APIs for the Modern Web

PHP Web Services

O'REILLY®

Lorna Jane Mitchell

PHP Web Services

Whether you're sharing data between two internal systems or building an API so users can access their data, this practical book provides everything you need to build web service APIs with PHP. Author Lorna Jane Mitchell uses code samples, real-world examples, and advice based on her extensive experience to guide you through the process—from the underlying theory to methods for making your service robust.

PHP is ideally suited for both consuming and creating web services. You'll learn how to use this language with JSON, XML, and other web service technologies.

- Explore HTTP, from the request/response cycle to its verbs, headers, and cookies
- Determine whether JSON or XML is the best data format for your application
- Get practical advice for working with RPC, SOAP, and RESTful services
- Use a variety of tools and techniques for debugging HTTP web services
- Choose the service that works best for your application, and learn how to make it robust
- Learn how to document your API—and how to design it to handle errors

Twitter: @oreillymedia
facebook.com/oreilly

US $14.99 CAN $15.99
ISBN: 978-1-449-35656-9

O'REILLY®
oreilly.com

PHP Web Services

Lorna Jane Mitchell

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

PHP Web Services

by Lorna Jane Mitchell

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Maria Gulick and Rachel Roumeliotis
Production Editor: Marisa LaFleur
Proofreader: Marisa LaFleur
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

April 2013: First Edition

Revision History for the First Edition:

2013-04-19: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449356569* for release details.

ISBN: 978-1-449-35656-9

[LSI]

Table of Contents

Preface

In this age, when it can sometimes seem like every system is connected to every other system, dealing with data has become a major ingredient in building the Web. Whether you will be delivering services or consuming them, web service is a key part of all modern, public-facing applications, and this book is here to help you navigate your way along the road ahead. We will cover the different styles of service—from RPC, to SOAP, to REST—and you will see how to devise great solutions using these existing approaches, as well as examples of APIs in the wild. Whether you're sharing data between two internal systems, using a service backend for a mobile application, or just plain building an API so that users can access their data, this book has you covered, from the technical sections on HTTP, JSON, and XML to the "big picture" areas such as creating a robust service.

Why did we pick PHP for this book? Well, PHP has always taken on the mission to "solve the web problem." Web services are very much part of that "problem" and PHP is ideally equipped to make your life easy, both when consuming external services and when creating your own. As a language, it runs on many platforms and is the technology behind more than half of the Web, so you can be sure that it will be widely available, wherever you are. This book does not adopt any particular frameworks; instead, it aims to give you the tools you will need to understand the topic as a whole and apply that knowledge to whichever frameworks, libraries, or other wrappers you choose to use.

The book walks you through everything you need to know in three broad sections. We begin by covering HTTP and the theory that goes with it, including detailed chapters on the request/response cycle, HTTP verbs and headers, and cookies. There are also chapters on JSON and XML: when to choose each data format, and how to handle them from within PHP. The second section aims to give very practical advice on working with RPC and SOAP services, with RESTful services, and on how to debug almost anything that works over HTTP, using a variety of tools and techniques. In the final section, we look at some of the wider issues surrounding the design of top-quality services, choosing what kind of service will work for your application, and determining how to make it robust. Another chapter is dedicated to handling errors and giving advice on why and

how to document your API. Whether you dip into the book as a reference for a specific project, or read it in order to find out more about this area of technology, there's something here to help you and your project to be successful. Enjoy!

Conventions Used in This Book

The following typographical conventions are used in this book:

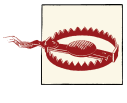
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*PHP Web Services* by Lorna Jane Mitchell (O'Reilly). Copyright 2013 Lorna Jane Mitchell, 978-1-449-35656-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/php-web-services*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

While this is quite a small book on the scale of things, a great many people gave their input to make it happen and they deserve to be acknowledged for the contributions they made.

Several people reviewed early drafts of the book from a technical standpoint and asked many difficult questions at a stage when there was scope for answering them. Thanks to Sean Coates, Jon Phillips, Michele Davis, and Chris Willcock for all their input.

My editors Maria Gulick and Rachel Roumeliotis have been patient and supportive throughout, something I'm sure gets tiring with such a large number of titles coming past at high speed. Their advice and support were invaluable, and I thank them for their gracious help. The rest of the O'Reilly staff have been rockstars also, in particular Josette Garcia, who always makes me believe, and the team that supports the tools I broke so regularly.

My wider "geek support network" has been at once encouraging and providers of practical help. Many people rescued me from my own code samples, gave advice where my own experience fell short, and pointed me to further reading on a variety of topics that made it into this book (and many others that did not). This was very much a hive effort and I consider myself lucky to be part of a community from which help can be requested and given so readily.

Finally, thanks are due to my mystified, but fantastically supportive, family and friends. Chief among these, of course, is my husband, Kevin, who served as cheerleader, proofreader, and head technical support consultant throughout this project and so many others.

CHAPTER 1
HTTP

HTTP stands for HyperText Transfer Protocol, and is the basis upon which the Web is built. Each HTTP transaction consists of a *request* and a *response*. The HTTP protocol itself is made up of many pieces: the URL at which the request was directed, the verb that was used, other headers and status codes, and of course, the body of the responses, which is what we usually see when we browse the Web in a browser.

When surfing the Web, ideally we experience a smooth journey between all the various places that we'd like to visit. However, this is in stark contrast to what is happening behind the scenes as we make that journey. As we go along, clicking on links or causing the browser to make requests for us, a series of little "steps" is taking place behind the scenes. Each step is made up of a request/response pair; the client (usually your browser or phone if you're surfing the Web) makes a request to the server, and the server processes the request and sends the response back. At every step along the way, the client makes a request and the server sends the response.

As an example, point a browser to *http://oreilly.com/* and you'll see a page that looks something like Figure 1-1; either the information desired can be found on the page, or the hyperlinks on that page direct us to journey onward for it.

Figure 1-1. O'Reilly home page

The web page arrives in the body of the HTTP response, but it tells only half of the story. The rest is elsewhere in the HTTP traffic. Consider the following examples.

Request header:

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.8 (KHTML, like Gecko)
Chrome/23.0.1246.0 Safari/537.8
Host: oreilly.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
```

Response header:

```
HTTP/1.1 200 OK
Date: Thu, 15 Nov 2012 09:36:05 GMT
Server: Apache
Last-Modified: Thu, 15 Nov 2012 08:35:04 GMT
Accept-Ranges: bytes
Content-Length: 80554
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Thu, 15 Nov 2012 13:36:05 GMT
Vary: Accept-Encoding
```

As you can see, there are plenty of other useful pieces of information being exchanged over HTTP that are not usually seen when using a browser. Understanding this separation between client and server, and the steps taken by the request and response pairs, is key to understanding HTTP and working with web services. Here's an example of what happens when we head to Google in search of kittens:

1. We make a request to *http://www.google.com/* and the response contains a Location header and a 301 status code sending us to a regional search page; for me that's *http://www.google.co.uk/*.
2. The browser follows the redirect instruction (without confirmation from the user, browsers follow redirects by default) and makes a request to *http://www.google.co.uk/* and recceives the page with the search box (for fun, view the source of this page. There's a lot going on!). We fill in the box and hit search.
3. We make a request to *https://www.google.co.uk/search?q=kittens* (plus a few other parameters) and get a response showing our search results.

In the story shown here, all the requests were made from the browser in response to a user's actions, although some occur behind the scenes, such as following redirects or requesting additional assets. All the assets for a page, such as images, stylesheets, and so on are all fetched using separate requests that are handled by a server. Any content that is loaded asynchronously (by JavaScript, for example) also creates more requests. When we work with APIs, we get closer to the requests and make them in a more deliberate manner, but the mechanisms are the same as those we use to make very basic web pages. If you're already making websites, then you already know all you need to make web services!

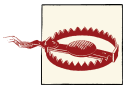
Clients and Servers

Earlier in this chapter we talked about a request and response between a client and a server. When we make websites with PHP, the PHP part is always the server. When using APIs, we build the server in PHP, but we can consume APIs from PHP as well. This is the point where things can get confusing. We can create either a client or a server in PHP, and requests and responses can be either incoming or outgoing—or both!

When we build a server, we follow patterns similar to the way that we build web pages. A request arrives, and we use PHP to figure out what was requested and craft the correct response. For example, if we built an API for customers so they could get updates on their orders programmatically, we would be building a server.

Using PHP to consume APIs means we are building a client. Our PHP application makes requests to external services over HTTP, and then uses the responses for its own purposes. An example of a client would be a page that fetches your most recent tweets and displays them.

It isn't unusual for an application to be *both* a client and a server, as shown in Figure 1-2. An application that accepts a request, and then calls out to other services to gather the information it needs to produce the response, is acting as both a client and a server.



When working on applications like this, take care with how you name variables involving the word "request" to avoid confusion!

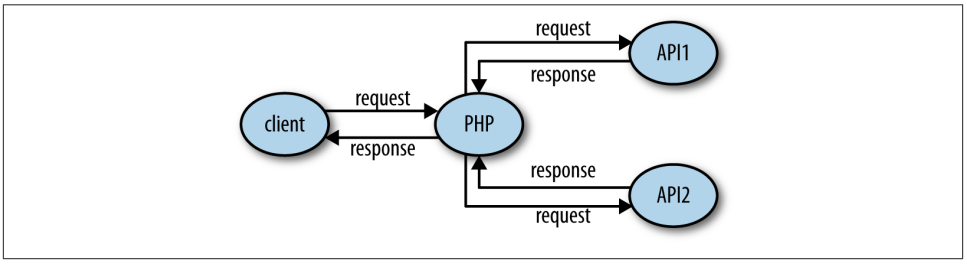


Figure 1-2. Web application acting as a server to the user, but also as a client to access other APIs

Making HTTP Requests

There are a few different ways to communicate over HTTP. In this section, three of them will be covered: Curl, tools in your browser, and PHP itself. The tool you choose depends entirely on your experience and on what it is that you're trying to achieve. We'll also look at tools for inspecting and debugging HTTP in Chapter 9.

The examples here use a site that is set up to log requests made to it, which is perfect for exploring how different API requests are seen by a server. To use it, visit the site and create a new "request bin." You will see the URL needed to make requests to and be redirected to a page showing the history of requests made to the bin. Another excellent way to try making different kinds of requests is to use the reserved endpoints (*http://example.com*, *http://example.net*, and *http://example.org*) established by the Internet Assigned Numbers Authority.

Curl

Curl is a command-line tool available on all platforms. It allows us to make any web request imaginable in any form, repeat those requests, and observe in detail exactly what information is exchanged between client and server. In fact, Curl produced the example output at the beginning of this chapter. It is a brilliant, quick tool for inspecting what's

going on with a web request, particularly when dealing with those outside the usual scope of a browser.

In its most basic form, a Curl request can be made like this (replace the URLs with your own):

```
curl http://requestb.in/example
```

We can control every aspect of the request to send; some of the most commonly used features are outlined here and used throughout this book to illustrate and test the various APIs shown.

If you've built websites before, you'll already know the difference between `GET` and `POST` requests from creating web forms. Changing between `GET`, `POST`, and other HTTP verbs using Curl is done with the `-X` switch, so a POST request can be specifically made by using the following:

```
curl -X POST http://requestb.in/example
```

To get more information from Curl than just the body response, there are a couple of useful switches. Try the `-v` switch since this will show everything: request headers, response headers, and response body in full! It splits the response up, though, sending the header information to STDERR and the body to STDOUT.

When the response is fairly large, it can be hard to find a particular piece of information while using Curl. To help with this, it is possible to combine Curl with other tools such as `less` or `grep`; however, Curl shows a progress output bar in normal operation, which is confusing to these other tools. To silence the progress bar, use the `-s` switch (but beware that it also silences Curl's errors). It can be helpful to use `-s` in combination with `-v` to create output that you can send to a pager such as `less` in order to examine it in detail, using a command like this:

```
curl -s -v http://requestb.in/example 2>&1 | less
```

The extra `2>&1` is there to send the STDERR output to STDOUT so that you'll see both headers and body; by default, only STDOUT would be visible to `less`.

Working with the Web in general, and APIs in particular, means working with data. Curl lets us do that in a few different ways. The simplest way is to send data along with a request in key/value pairs—exactly as when a form is submitted on the Web—which uses the `-d` switch. The switch is used as many times as there are fields to include:

```
curl -X POST http://requestb.in/example -d name="Lorna" -d email="lorna@exam-
ple.com" -d message="this HTTP stuff is rather excellent"
```

APIs accept their data in different formats; sometimes the data cannot be `POST`ed as a form, but must be created in JSON or XML format, for example. In such instances, the entire body of a request can be assembled in a file and passed to Curl. Inspect the previous request, and you will see that the body of it is sent as:

```
name=Lorna&email=lorna@example.com&message=this HTTP stuff is excellent
```

Instead of sending the data as key/value pairs on the command line, it can be placed into a file called *data.txt* (for example). This file can then be supplied each time the request is made. This technique is especially useful for avoiding very long command lines when working with lots of fields, and when sending non-form data, such as JSON or XML. To use the contents of a file as the body of a request, we give the filename prepended with an `@` symbol as a single `-d` switch to Curl:

```
curl -X POST http://requestb.in/example -d @data.txt
```

Working with the extended features of HTTP requires the ability to work with various headers. Curl allows sending of any desired header (this is why, from a security standpoint, the header can never be trusted!) by using the `-H` switch, followed by the full header to send. The command to set the Accept header to ask for an HTML response becomes:

```
curl -H "Accept: text/html" http://requestb.in/example
```

Before moving on from Curl to some other tools, let's take a look at one more feature: how to handle cookies. Cookies will be covered in more detail in a later chapter, but for now it is just important to know that cookies are stored by the client and sent with requests, and that new cookies may be received with each response. Browsers send cookies with requests as default behavior, but in Curl we need to do this manually by asking Curl to store the cookies in a response and then use them on the next request. The file that stores the cookies is called the "cookie jar"; clearly, even HTTP geeks have a sense of humor.

To receive and store cookies from one request:

```
curl -c cookiejar.txt http://requestb.in/example
```

At this point, *cookiejar.txt* can be amended in any way you see fit (again, never trust information that came from outside the application!), and then sent to the server with the next request you make. To do this, use the `-b` switch and specify the file to find the cookies in:

```
curl -b cookiejar.txt http://requestb.in/example
```

To capture cookies and resend them with each request, use both `-b` and `-c` switches, referring to the same *cookiejar* file. This way, all incoming cookies are captured and sent to a file, and then sent back to the server on any subsequent request, behaving just as they do in a browser.

Browser Tools

All the newest versions of the modern browsers (Chrome, Firefox, Opera, Safari, Internet Explorer) have built-in tools or available plug-ins for helping to inspect the HTTP that's being transferred, and for simple services you may find that your browser's tools

are an approachable way to work with an API. These tools vary between browsers and are constantly updating, but here are a few favorites to give you an idea.

In Firefox, this functionality is provided by the Developer Toolbar and various plug-ins. Many web developers are familiar with FireBug, which does have some helpful tools, but there is another tool that is built specifically to show you all the headers for all the requests made by your browser: LiveHTTPHeaders. Using this, we can observe full details of each request, as seen in Figure 1-3.

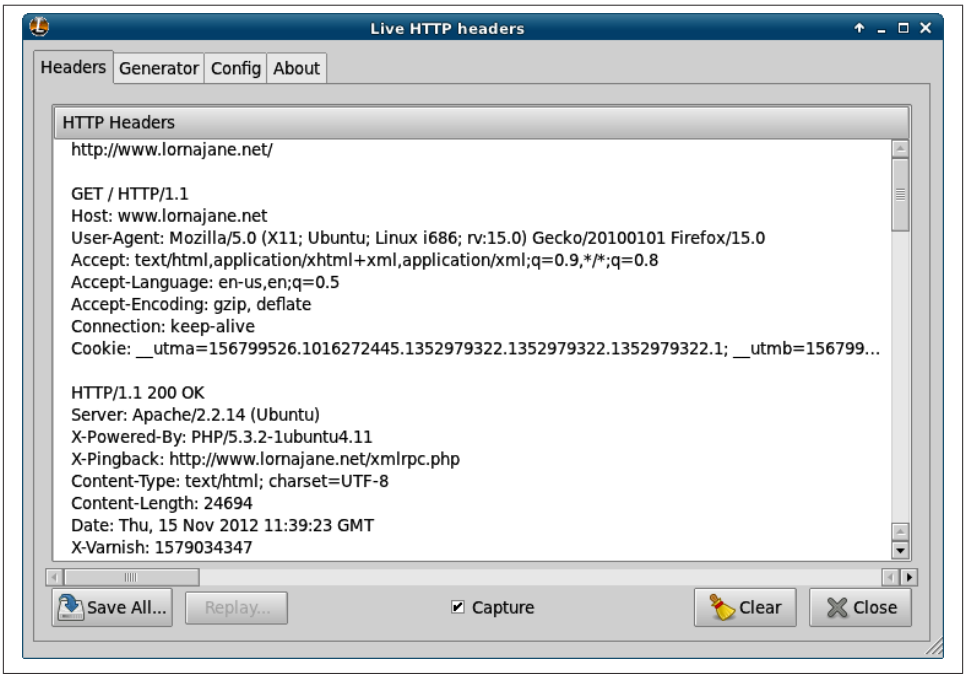


Figure 1-3. LiveHTTPHeaders showing HTTP details

All browsers offer some way to inspect and change the cookies being used for requests to a particular site. In Chrome, for example, this functionality is offered by an extension called "Edit This Cookie," and other similar extentions. This shows existing cookies and lets you edit and delete them—and also allows you to add new cookies. Take a look at the tools in your favorite browser and see the cookies sent by the sites you visit the most.

Sometimes, additional headers need to be added to a request, such as when sending authentication headers, or specific headers to indicate to the service that we want some extra debugging. Often, Curl is the right tool for this job, but it's also possible to add the headers into your browser. Different browsers have different tools, but for Chrome try an extension called ModHeader, seen in Figure 1-4.

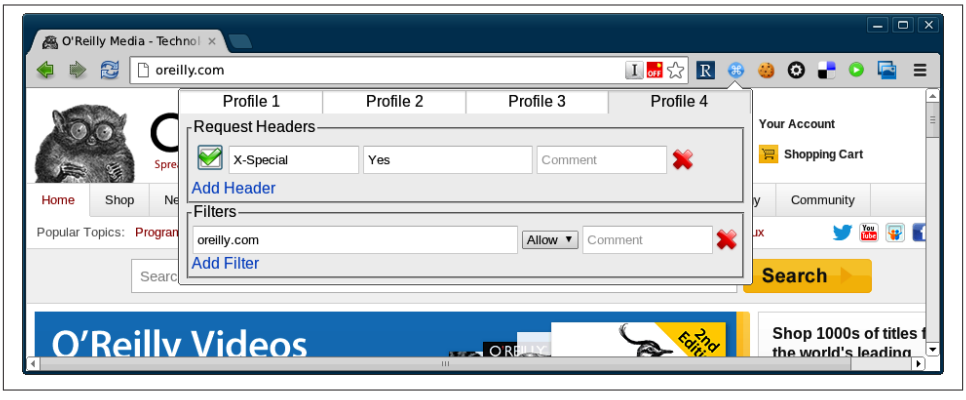


Figure 1-4. The ModHeader plug-in in Chrome

PHP

Unsurprisingly, there is more than one way to handle HTTP requests using PHP, and each of the frameworks will also offer their own additions. This section focuses on plain PHP and looks at three different ways to work with APIs: using the built-in Curl extension for PHP, using the `pecl_http` extension, and making HTTP calls using PHP's stream handling.

Earlier in this chapter, we discussed a command-line tool called Curl (see "Curl" on page 4). PHP has its own wrappers for Curl, so we can use the same tool from within PHP. A simple `GET` request looks like this:

```
<?php

$url = "http://oreilly.com";
$ch = curl_init($url);

curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

The previous example is the simplest form, setting the URL, making a request to its location (by default this is a `GET` request), and capturing the output. Notice the use of `curl_setopt()`; this function is used to set many different options on Curl handles and it has excellent and comprehensive documentation on *http://php.net*. In this example, it is used to set the `CURLOPT_RETURNTRANSFER` option to *true*, which causes Curl to *return* the results of the HTTP request rather than *output* them. In most cases, this option should be used to capture the response rather than letting PHP echo it as it happens.

We can use this extension to make all kinds of HTTP requests, including sending custom headers, sending body data, and using different verbs to make our request. Take a look

at this example, which sends some form fields and a `Content-Type` header with the `POST` request:

```
<?php

$url = "http://requestb.in/example";
$data = array("name" => "Lorna", "email" => "lorna@example.com");

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));

curl_setopt($ch, CURLOPT_HTTPHEADER,
    array('Content-Type: application/json')
);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

Again, `curl_setopt()` is used to control the various aspects of the request we send. Here, a `POST` request is made by setting the `CURLOPT_POST` option to 1, and passing the data we want to send as an array to the `CURLOPT_POSTFIELDS` option. We also set a `Content-Type` header, which indicates to the server what format the body data is in; the various headers are covered in more detail in Chapter 3.

The PHP Curl extension isn't the easiest interface to use, although it does have the advantage of being reliably available. A great alternative if you control your own platforms is to add the `pecl_http` extension from PECL. This offers a much more intuitive way of working and has both function and object-oriented interfaces. For example, here's the previous example, this time using `pecl_http`:

```
<?php

$url = "http://requestb.in/example";

$data = array("name" => "Lorna", "email" => "lorna@example.com");

$request = new HTTPRequest($url, HTTP_METH_POST);
$request->setPostFields($data);
$request->setHeaders(array("Content-Type" => "application/json"));

$request->send();
$result = $request->getResponseBody();
```

This extension works more elegantly by creating an `HTTPRequest` object, and then working with the properties on that object, before calling its `send()` method. Once the request has been sent, the body of the response is fetched by calling the `getResponseBody()` method.

Finally, let's look at one more way of making HTTP requests from PHP: using PHP's stream-handling abilities with the file functions. In its simplest form, this means that, if `allow_url_fopen` is enabled (see the PHP manual), it is possible to make a `GET` request using `file_get_contents()`:

```
<?php

$result = file_get_contents("http://oreilly.com");
```

We can take advantage of the fact that PHP can handle a variety of different protocols (HTTP, FTP, SSL, and more) and files using streams. The simple `GET` requests are easy, but what about something more complicated? Here is an example that makes the same `POST` request with headers, illustrating how to use various aspects of the streams functionality:

```
<?php

$url = "http://requestb.in/example";
$data = array("name" => "Lorna", "email" => "lorna@example.com");

$context = stream_context_create(array(
    'http' => array(
        'method' => 'POST',
        'header' => array('Accept: application/json',
            'Content-Type: application/x-www-form-urlencoded'),
        'content' => http_build_query($data)
    )
));

$result = file_get_contents($url, false, $context);
```

Options are set as part of the *context* that we create to dictate how the request should work. Then, when PHP opens the stream, it uses the information supplied to determine how to handle the stream correctly—including sending the given data and setting the correct headers.

As you can see, there are a few different options for dealing with HTTP, both from PHP and the command line, and you'll see all of them used throughout this book. These approaches are all aimed at "vanilla" PHP, but if you're working with a framework, it will likely offer some functionality along the same lines; all the frameworks will be wrapping one of these methods so it will be useful to have a good grasp of what is happening underneath the wrappings. After trying out the various examples, it's common to pick one that you will work with more than the others; they can all do the job, so the one you pick is a result of both personal preference and which tools are available (or can be made available) on your platform.

CHAPTER 2

HTTP Verbs

HTTP verbs such as `GET` and `POST` let us send our intention along with the URL so we can instruct the server what to do with it. Web requests are more than just a series of addresses, and verbs contribute to the rich fabric of the journey.

I mentioned `GET` and `POST` because it's very likely you're already familiar with those. There are many verbs that *can* be used with HTTP—in fact, we can even invent our own—but we'll get to that later in the chapter (see "Using Other HTTP Verbs" on page 15). First, let's revisit `GET` and `POST` in some detail, looking at when to use each one and what the differences are between them.

Making GET Requests

URLs used with `GET` can be bookmarked, they can be called as many times as needed, and the request should not affect change to the data it accesses. A great example of using a `GET` request when filling in a web form is when using a search form, which should always use `GET`. Searches can be repeated safely, and the URLs can be shared.

Consider the simple web form in Figure 2-1, which allows users to state which category of results they'd like and how many results to show. The code for displaying the form and the (placeholder) search results on the page could be something like this:

```
<?php

if(empty($_GET)) {

?>

<form name="search" method="get">
    Category:
    <select name="category">
        <option value="entertainment">Entertainment</option>
        <option value="sport">Sport</option>
```

```
        <option value="technology">Technology</option>
    </select> <br />

    Rows per page: <select name="rows">
        <option value="10">10</option>
        <option value="20">20</option>
        <option value="50">50</option>
    </select> <br />

    <input type="submit" value="Search" />
</form>

<?php
} else {
    echo "Wonderfully filtered search results";
}
```

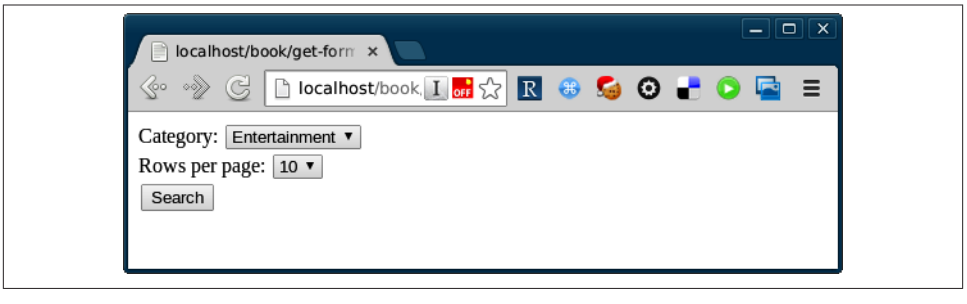


Figure 2-1. An example search form

You can see that PHP simply checks if it has been given some search criteria (or indeed any data in the `$_GET` superglobal) and if not, it displays the empty form. If there was data, then it would process it (although probably in a more interesting way than this trivial example does). The data gets submitted on the URL when the form is filled in (`GET` requests typically have no body data), resulting in a URL like this:

```
http://localhost/book/get-form-page.php?category=technology&rows=20
```

The previous example showed how PHP responds to a `GET` request, but how does it make one? Well, as discussed in Chapter 1, there are many ways to approach this. For a very quick solution, and a useful approach to use when working with `GET` requests in particular, use PHP's stream handling to create the complete request to send:

```
<?php

$url = 'http://localhost/book/get-form-page.php';
$data = array("category" => "technology", "rows" => 20);

$get_addr = $url . '?' . http_build_query($data);
```

```
$page = file_get_contents($get_addr);
echo $page;
```

In a real system, it is prudent to be cautious of the data coming in from external APIs; it is best to filter the contents of `$page` before outputting it or using it anywhere else. As an alternative to using PHP's stream features, you could use whatever functionality your existing frameworks or libraries offer, or make use of the Curl extension that is built in to PHP. Using Curl, our code would instead look like this:

```
<?php

$url = 'http://localhost/book/get-form-page.php';
$data = array("category" => "technology", "rows" => 20);

$get_addr = $url . '?' . http_build_query($data);
$ch = curl_init($get_addr);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
$page = curl_exec($ch);
echo $page;
```

Either of these approaches work well when you want to fetch data into your PHP script from an external API or page. The examples here show web pages, but they apply when working with HTML, XML, JSON, or anything else.

Making POST Requests

In contrast to `GET` requests, a `POST` request is one that does cause change on the server that handles the request. These requests shouldn't be repeated or bookmarked, which is why your browser warns you when it is resubmitting data. Let's use a `POST` form when the request changes data on the server side. Figure 2-2, for example, involves updating a bit of user profile information.

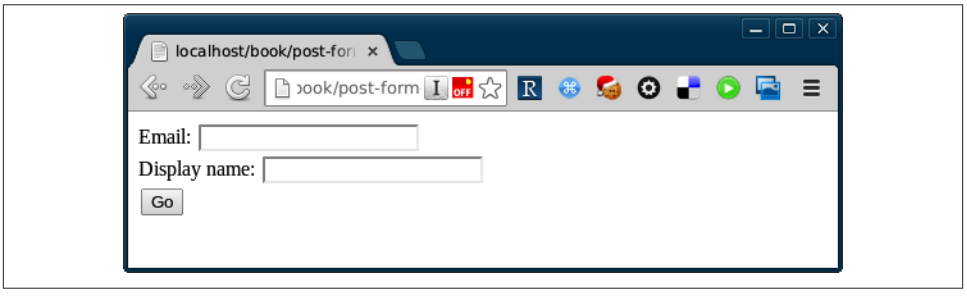


Figure 2-2. Simple form that updates data, sending content via a POST request

When a form is submitted via `GET`, we can see the variables being sent on the URL. With `POST`, however, the data goes into the *body* of the request, and the `Content-Type` header

denotes what kind of data can be found in the body. When we fill in the form in Figure 2-2, the request looks like this:

```
POST /book/post-form-page.php HTTP/1.1
Host: localhost
Content-Length: 48
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

email=lorna%40example.com&display_name=LornaJane
```

In this example, you can see the data in the body, with the `Content-Type` and `Content-Length` headers set appropriately so a client could decode the response (more about content negotiation in Chapter 3). PHP knows how to handle form data, so it can parse this out and place the fields into `$_POST`, so it will be ready for use in the script. Here is the code behind this page, showing the form without any incoming data; if data existed, it would be displayed:

```
<?php

if(empty($_POST)) {

?>

<form name="user" method="post">
    Email:
    <input type="text" length="60" name="email" /><br />

    Display name:
    <input type="text" length="60" name="display_name" /><br />

    <input type="submit" value="Go" />
</form>

<?php
} else {
    echo "New user email: " . filter_input(INPUT_POST,
        "email", FILTER_VALIDATE_EMAIL);
}
```

It is very common to build PHP forms and parse data in this way, but when handling HTTP requests, it is also important to consider how the requests can be made and responded to. This is not dissimilar to the way that `GET` requests are made. For example, to `POST` data to this form using streams (as in "Making GET Requests" on page 11), the same basic approach can be used, but some *context* should be added to the stream, so it will know which methods, headers, and verbs to use:

```
<?php

$url = 'http://localhost/book/post-form-page.php';
$data = array("email" => "lorna@example.com", "display_name" => "LornaJane");
$options = array(
    "http" => array(
        "method"  => "POST",
        "header"  => "Content-Type: application/x-www-form-urlencoded",
        "content" => http_build_query($data)
    )
);

$page = file_get_contents($url, false, stream_context_create($options));
echo $page;
```

When `POST` data is sent to the page created, the data sent appears in the output rather than in the form, so it shows "New user email: *lorna@example.com*." This code looks very similar to the previous streams example, but this example uses `stream_context_create()` to add some additional information to the stream.

You can see that we added the body content as a simple string, formatted it as a URL using `http_build_query()`, and indicated which content type the body is. This means that other data formats can very easily be sent by formatting the strings correctly and setting the headers.

Here is an example that does the exact same thing, but uses the `pecl_http` extension:

```
<?php

$url = 'http://localhost/book/post-form-page.php';
$data = array("email" => "lorna@example.com", "display_name" => "LornaJane");

$request = new HttpRequest($url, HTTP_METH_POST);
$request->setPostFields($data);
$request->send();
$page = $request->getResponseBody();
echo $page;
```

Because this is a `POST` request, PHP assumes that a form is being posted; but different `Content-Type` headers can be set if appropriate, and another format of string data can be sent. This approach is illustrated in many different ways as this book progresses. When working with non-standard verbs (as seen in the next section) or with data that isn't from a form post, it isn't possible to access the data by grabbing it from `$_POST`. Instead, PHP's own stream of raw body data can be accessed at *php://input*.

Using Other HTTP Verbs

There are many specifications relating to HTTP, as well as protocols based upon it, and between them they define a wide selection of verbs that can be used with HTTP. Even

better, there is always room to invent new HTTP verbs; so long as your client and server both know how to handle a new verb, it is valid to use it. However, be aware that not all elements of network infrastructure between these two points will necessarily know how to handle every verb. Some pieces of network infrastructure do not support `PATCH`, for example, or the verbs used by the WebDAV protocol. When working with APIs, particularly RESTful ones, it is normal to make use of two additional verbs: `PUT` and `DELETE`. REST is covered in detail in Chapter 8, but for now it is useful to examine some examples of how to use these less common verbs in applications.

The simplest of these two is `DELETE`, because it doesn't have any body data associated with it. It is possible to see what kind of request was made to a PHP script acting as a server by inspecting the `$_SERVER["REQUEST_METHOD"]` value, which indicates which verb was used in the request.

To make the request from PHP, it is necessary to set the verb and then make the request as normal. Here's an example using the Curl extension:

```
<?php

$url = 'http://localhost/book/example-delete.php';

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
curl_exec($ch);
```

This example simply issues a request to the `$url` shown using a `DELETE` verb.

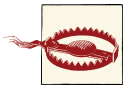
Using `PUT` is slightly more involved because, like `POST`, it can be accompanied by data and the data can be in a variety of formats. In "Making POST Requests" on page 13, I mentioned that for incoming form data, PHP reads form-encoded values for `POST` and creates a `$_POST` array for us. There is no equivalent to `$_PUT` superglobal, but we can still make use of the *php://input* stream to inspect the body data of the request to which the script is sending a response at that time.

When using PHP to respond to `PUT` requests, the code runs along the lines of this example:

```
<?php

if($_SERVER['REQUEST_METHOD'] == "PUT") {
    $data = array();
    $incoming = file_get_contents("php://input");
    parse_str($incoming, $data);
    echo "New user email: " . filter_var($data["email"], FILTER_VALIDATE_EMAIL);
} else {
    echo "um?";
}
```

This example inspects the `$_SERVER` superglobal to see which verb was used, and then responds accordingly. The data coming into this example is form style, meaning it uses `file_get_contents()` to grab all the body data, then `parse_str()` to decode it.



Be careful with `parse_str()`—if the second argument is omitted, the variables will be extracted as local variables, rather than contained in an array.

In order to use PHP to make a request that the previous script can handle, it is necessary to create the contents of the body of the request and specify that it is a `PUT` request. Below is an example built using the `pecl_http` extension:

```
<?php

$url = 'http://localhost/book/put-form-page.php';
$data = array("email" => "lorna@example.com", "display_name" => "LornaJane");

$request = new HttpRequest($url, HTTP_METH_PUT);
$request->setHeaders(array(
    "Content-Type" => "application/x-www-form-urlencoded"));
$request->setPutData(http_build_query($data));
$request->send();
$page = $request->getResponseBody();
echo $page;
```

The `PUT` verb is specified in this example, and the correct header for the form-encoded data is set. We dictate the data to `PUT` (manually building the form elements into a string) and then send the request. We will discuss more about other data formats in Chapter 5 and Chapter 6, which cover JSON and XML specifically, but the basic principles of preparing the data and setting the `Content-Type` header accordingly still stand.

Armed with this knowledge of how to handle `GET`, `POST`, `DELETE`, and `PUT` verbs, we are able to work with many different kinds of API acting as both a client and as a server. When using other verbs, either those that already exist as part of the HTTP spec or those that are custom to your application, you can use the approaches described here for `PUT` and `DELETE`.

CHAPTER 3
Headers

So far, we've seen various presentations of the HTTP format, and examined the idea that there is a lot more information being transferred in web requests and responses than what appears in the body of the response. The body is certainly the most important bit, and often is the meatiest, but the headers provide key pieces of information for both requests and responses, which allow the client and the server to communicate effectively. If you think of the body of the request as a birthday card with a check inside it, then the headers are the address, postmark, and perhaps the "do not open until…" instruction on the outside (see Figure 3-1).

This additional information gets the body data to where it needs to go and instructs the target on what to do with it when it gets there.

Figure 3-1. Envelope with stamp, address, and postmark

Request and Response Headers

Many of the headers you see in HTTP make sense in both requests and responses. Others might be specific to *either* a request *or* a response. Here's a sample set of real request and response headers from when I request my own site from a browser (I'm using Chrome).

Request headers:

```
GET / HTTP/1.1
Host: www.lornajane.net
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.19 (KHTML, like
Gecko) Chrome/25.0.1323.1 Safari/537.19
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,+;q=0.3
```

Response headers:

```
HTTP/1.1 200 OK
Server: Apache/2.2.14 (Ubuntu)
X-Powered-By: PHP/5.3.2-1ubuntu4.11
X-Pingback: http://www.lornajane.net/xmlrpc.php
Last-Modified: Thu, 06 Dec 2012 14:46:05 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Length: 25279
Date: Thu, 06 Dec 2012 14:46:05 GMT
X-Varnish: 2051611642
Age: 0
Via: 1.1 varnish
Connection: keep-alive
```

Here, you see `Content-Type` set in the body of the response, but it would also be used when `POST`ing data with a request. Such multiple-use headers are called *entity headers* and relate to the body being sent with the HTTP request or response. Specific headers that are sent with requests are `User-Agent`, `Accept`, `Authorization`, and `Cookie`, and `Set-Cookie` is returned with responses.

Common HTTP Headers

The previous examples showed off a selection of common headers, while the next sections move on to take a look at the headers most often encountered when working with APIs. The following examples show how to send and receive various types of headers from PHP so that you can handle headers correctly in your own applications.

User-Agent

The `User-Agent` header gives information about the client making the HTTP request and usually includes information about the software client. Take a look at the header here:

```
User-Agent Mozilla/5.0 (Linux; U; Android 2.3.4; en-gb; SonyEricssonSK17i Build/
4.0.2.A.0.62) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/
533.1
```

What device do you think made this request? You would probably guess that it was my Sony Ericsson Android phone…and perhaps you would be right. Or perhaps I used a Curl command:

```
curl -H "User-Agent: Mozilla/5.0 (Linux; U; Android 2.3.4; en-gb; SonyEricss-
sonSK17i Build/4.0.2.A.0.62) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0
Mobile Safari/533.1" http://requestb.in/example
```

We simply have no way of knowing, when a request is received with a `User-Agent` like this, if it *really* came from an Android phone, or if it came from something else *pretending* to be an Android phone. This information can be used to customize the response we send—after all, if someone wants to pretend to be a tiny Android phone, then it is reasonable to respond with the content that would normally be sent to this phone. It does mean, however, that the `User-Agent` header cannot be relied upon for anything more important, such as setting a custom header and using it as a means of authenticating users. Just like any other incoming data, it is wide open to abuse and must be treated with suspicion.

In PHP, it is possible both to parse and to send the `User-Agent` header, as suits the task at hand. Here's an example of sending the header using streams:

```
<?php

$url = 'http://localhost/book/user-agent.php';
$options = array(
    "http" => array(
        "header"  => "User-Agent: Advanced HTTP Magic Client"
    )
);

$page = file_get_contents($url, false , stream_context_create($options));
echo $page;
```

We can set any arbitrary headers we desire when making requests, all using the same approach. Similarly, headers can be retrieved using PHP by implementing the same approach throughout. The data of interest here can all be found in `$_SERVER`, and in this case it is possible to inspect `$_SERVER["HTTP_USER_AGENT"]` to see what the `User-Agent` header was set to.

To illustrate, here's a simple script:

```
<?php

echo "This request made by: "
    . filter_var($_SERVER['HTTP_USER_AGENT'], FILTER_SANITIZE_STRING);
```

It's common when developing content for the mobile web to use headers such as `User-Agent` in combination with WURFL to detect what capabilities the consuming device has, and adapt the content accordingly. With APIs, however, it is better to expect the clients to use other headers so they can take responsibility for requesting the correct content types, rather than allowing the decision to be made centrally.

Headers for Content Negotiation

Commonly, the `Content-Type` header is used to describe what format the data being delivered in the body of a request or a response is in; this allows the target to understand how to decode this content. Its sister header, `Accept`, allows the client to indicate what kind of content is *acceptable*, which is another way of allowing the client to specify what kind of content it actually knows how to handle. As seen in the earlier example showing headers, here's the `Accept` header Google Chrome usually sends:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

To read an `Accept` header, consider each of the comma-separated values as an individual entity. This client has stated a preference for (in order):

- text/html
- application/xhtml+xml
- application/xml
- `*/*`

This means that if any of these formats are supplied, the client will understand our meaning. The second two entries, however, include some additional information: the `q` value. This is an indication of how much a particular option is preferred, where the default value is `q=1`.

Here, Chrome claims to be able to handle a content type of `*/*`. The asterisks are wildcards, meaning it thinks it can handle any format that could possibly exist—which seems unlikely. If an imaginary format is implemented that both our client and server understand, for example, Chrome won't know how to parse it, so `*/*` is misleading.

Using the `Accept` and `Content-Type` headers together to describe what can be understood by the client, and what was actually sent, is called "Content Negotiation." Using the headers to negotiate the usable formats means that meta-information is not tangled up with actual data as it would be when sending both kinds of parameters with the body or URL of the request. Including the headers is generally a better approach.

We can negotiate more than just content, too. The earlier example contained these lines:

```
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,+;q=0.3
```

These headers show other kinds of negotiation, such as declaring what encoding the client supports, which languages are preferred, and which character sets can be used. This enables decisions to be made about how to format the response in various ways, and how to determine which formats are appropriate for the consuming device.

Parsing an Accept header

Let's start by looking at how to parse an `Accept` header correctly. All `Accept` headers have a comma-separated list of values, and some include a `q` value that indicates their level of preference. If the `q` value isn't included for an entry, it can be assumed that `q=1` for that entry. Using the `Accept` header from my browser again, I can parse it by taking all the segments, working out their preferences, and then sorting them appropriately. Here's an example function that returns an array of supported formats in order of preference:

```
<?php

function parseAcceptHeader() {
    $hdr = $_SERVER['HTTP_ACCEPT'];
    $accept = array();
    foreach (preg_split('/\s*,\s*/', $hdr) as $i => $term) {
        $o = new \stdclass;
        $o->pos = $i;
        if (preg_match(",^(\S+)\s*;\s*(?:q|level)=([0-9\.]+),i", $term, $M)) {
            $o->type = $M[1];
            $o->q = (double)$M[2];
        } else {
            $o->type = $term;
            $o->q = 1;
        }
        $accept[] = $o;
    }
    usort($accept, function ($a, $b) {
        /* first tier: highest q factor wins */
        $diff = $b->q - $a->q;
        if ($diff > 0) {
            $diff = 1;
        } else if ($diff < 0) {
            $diff = -1;
        } else {
            /* tie-breaker: first listed item wins */
            $diff = $a->pos - $b->pos;
        }
        return $diff;
    });
```

```
    $accept_data = array();
    foreach ($accept as $a) {
        $accept_data[$a->type] = $a->type;
    }
    return $accept_data;
}
```

The headers sent by your browser may differ slightly and result in different output when you try the previous code snippet.

When using the `Accept` header sent by my browser, I see the following output:

```
array(4) {
  ["text/html"]=>
  string(9) "text/html"
  ["application/xhtml+xml"]=>
  string(21) "application/xhtml+xml"
  ["application/xml"]=>
  string(15) "application/xml"
  ["*/*"]=>
  string(3) "*/*"
}
```

We can use this information to work out which format it would be best to send the data back in. For example, here's a simple script that calls the `parseAcceptHeader()` function, then works through the formats to determine which it can support, and sends that information:

```
<?php

$data = array ("greeting" => "hello", "name" => "Lorna");

$accepted_formats = parseAcceptHeader();
$supported_formats = array("application/json", "text/html");
foreach($accepted_formats as $format) {
    if(in_array($format, $supported_formats)) {
        // yay, use this format
        break;
    }
}

switch($format) {
    case "application/json":
        header("Content-Type: application/json");
        $output = json_encode($data);
        break;
    case "text/html":
    default:
        $output = "<p>" . implode(',', $data) . "</p>";
```

```
            break;
    }

    echo $output;
```

There are many, many ways to parse the `Accept` header (and the same techniques apply to the `Accept-Language`, `Accept-Encoding`, and `Accept-Charset` headers), but it is vital to do so correctly. The importance of `Accept` header parsing can be seen in Chris Shiflett's blog post, The Accept Header; the `parseAcceptHeader()` example shown previously came mostly from the comments on this post. You might use this approach, an existing library such as the PHP mimeparse port, a solution you build yourself, or one offered by your framework. Whichever you choose, make sure that it parses these headers correctly, rather than using a string match or something similar.

Demonstrating Accept headers with Curl

Using Curl from the command line, here are some examples of how to call exactly the same URL by setting different `Accept` headers and seeing different responses:

```
curl http://localhost/book/hello.php
hello,Lorna

curl -H "Accept: application/json" http://localhost/book/hello.php
{"greeting":"hello","name":"Lorna"}

curl -H "Accept: text/html;q=0.5,application/json"
http://localhost/book/hello.php
{"greeting":"hello","name":"Lorna"}
```

To make these requests from PHP rather than from Curl, it is possible to simply set the desired headers as the request is made. Here's an example that uses PHP's `curl` extension to make the same request as the previous example:

```
<?php

$url = "http://localhost/book/hello.php";

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_HEADER, array(
    "Accept: text/html;q=0.5,application/json",
));
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
echo $response;
curl_close($ch);
```

The number of headers you need to support in your application will vary. It is common and recommended to offer various content types such as JSON, XML, or even plain text. The selection of supported encodings, languages, and character sets will depend entirely on your application and users' needs. If you do introduce support for variable content types, however, this is the best way to do it.

Securing Requests with the Authorization Header

Headers can provide information that allows an application to identify users. Again, keeping this type of information separate from the application data makes things simpler and, often, more secure. The key thing to remember when working on user security for APIs is that everything you already know about how to secure a website *applies to web services*. There's no need for anything new or inventive, and in fact I've seen some mistakes made because new wheels were invented instead of existing standards being embraced.

HTTP basic authentication

One of the simplest ways to secure a web page is to use HTTP basic authentication. This means that an encoded version of the user's credentials is sent in the Authorization header with every request. The underlying mechanics of this approach are simple: the client is given a username and password, and they do the following:

1. Arrange the username and password into the format `username:password`.
2. Base64 encode the result.
3. Send it in the header, like this: `Authorization: Basic` *`base64-encoded string`*.
4. Since tokens are sent in plain text, HTTPS should be used throughout.

We can either follow the steps here and manually create the correct header to send, or we can use the built-in features of our toolchain. Here's PHP's `curl` extension making a request to a page protected by basic authentication:

```
<?php

$url = "http://localhost/book/basic-auth.php";

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC ) ;
curl_setopt($ch, CURLOPT_USERPWD, "user:pass");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
echo $response;
curl_close($ch);
```

In PHP, these details can be found on the `$_SERVER` superglobal. When basic authentication is in use, the username and password supplied by the user can be found in `$_SERVER["PHP_AUTH_USER"]` and `$_SERVER["PHP_AUTH_PASSWORD"]`, respectively. When a request is made without credentials, or with invalid credentials, a 401 Unauthorized status code can be sent to tell the client why the server is not sending him the content requested.

OAuth

Another alternative for securing web services, especially when you have a third party consumer accessing data that belongs to a user, is OAuth. OAuth sets up a standard way for a consumer to gain access to anoher user's data that is held by a provider with whom the user already has a relationship, without the user giving away her password. The user visits the main provider's site to verify her identity and grant access to the consumer, and can also revoke that access at any time. Using this approach, the provider can distinguish between requests made by the user and requests made by something or someone else on behalf of the user.

The OAuth approach is beyond the scope of this book (*Getting Started with OAuth 2.0* [O'Reilly] is an excellent reference), but it does make use of the Authorization header and is widely used with APIs, so it is well worth a mention.

Custom Headers

As with almost every aspect of HTTP, the headers that can be used aren't set in stone. It is possible to invent new headers if there's more information to convey for which there isn't a header. Headers that aren't "official" can be used, but they should be prefixed with `X-`.

A good example, often seen on the Web, is when a tool such as Varnish has been involved in serving a response, and it adds its own headers. I have Varnish installed in front of my own site, and when I request it, I see:

```
HTTP/1.1 302 Found
Server: Apache/2.2.14 (Ubuntu)
Location: http://www.lornajane.net/
Content-Type: text/html; charset=iso-8859-1
Content-Length: 288
Date: Tue, 11 Dec 2012 15:53:46 GMT
X-Varnish: 119643096 119643059
Age: 5
Via: 1.1 varnish
Connection: keep-alive
```

That additional `X-Varnish` header shows me that Varnish served the request. It isn't an official header, but these `X-*` headers are used to denote all kinds of things in APIs and on the Web. A great example comes from GitHub. Here's what happens when I make a request to fetch a list of the repositories associated with my user account:

```
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 11 Dec 2012 16:01:00 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Status: 200 OK
X-Content-Type-Options: nosniff
```

```
Cache-Control: public, max-age=60, s-maxage=60
X-GitHub-Media-Type: github.beta
X-RateLimit-Limit: 60
Content-Length: 106586
Last-Modified: Sat, 01 Dec 2012 11:23:32 GMT
Vary: Accept
X-RateLimit-Remaining: 59
ETag: "8c0bde8e577f52c7f68de5d7099e041b"
```

There are a few custom headers in this example but the `X-RateLimit-*` headers are particularly worth noting, which check whether too many requests are being made. Using custom headers like these, any additional data can be transferred between client and server that isn't part of the body data, which means all parties can stay "on the same page" with the data exchange.

CHAPTER 4
Cookies

The HTTP protocol is *stateless*. This means that every request made must include all the information needed in order for the web server to serve the correct response. At least, in theory! In practice, that isn't how we experience the Web as users. As we browse around a shopping site, the website "remembers" which products we already viewed and which we placed in our basket—we experience our journeys on the Web as connected experiences.

So how does this work? Additional information is being saved and sent with our web requests through the use of *cookies*. Cookies are just key/value pairs; simple variables that can be stored on the client and sent back to us with future requests. A user's choice of theme or accessibility settings could be stored, or a cookie could be dropped to record something as simple as whether the user has visited the site before, or dismissed a particular alert message that was shown.

Cookie Mechanics

This isn't the moment where I tell you how to bake cookies, although the instructions do read a little bit like a recipe. What happens when we work with cookies goes something like this (see Figure 4-1):

1. A request arrives from the client, without cookies.
2. Send the response, including cookie(s).
3. The next request arrives. Since cookies were already sent, they will be sent back to us in these later requests.
4. Send the next response, also with cookies (either changed or unchanged).
5. Steps 3–4 are repeated indefinitely.

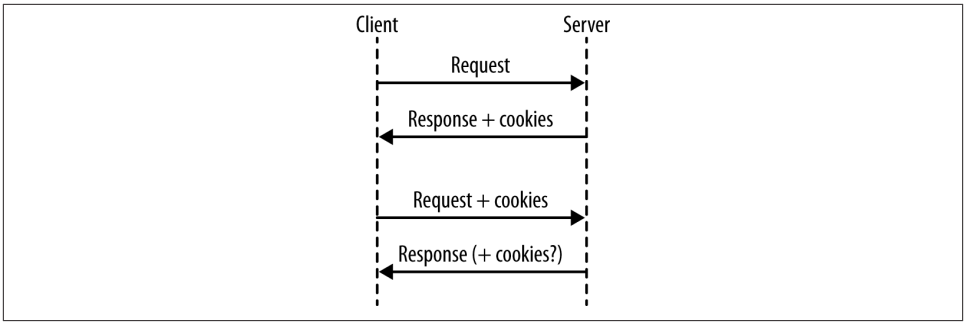


Figure 4-1. Cookies exchanged in a series of requests

The main thing to remember is that, for a first visit from a new client (or someone who clears their cookies), there will be no cookies, so it is not possible to rely on them being present. This is easy to miss in testing unless you consciously make the effort to also test the case in which a user arrives without cookies; by default, your browser will keep sending the cookies.

Another thing to note is that cookies are only sent back with subsequent requests *by convention*; not all clients will do this automatically. Once a cookie is received by a client, even if it isn't sent again in any later responses, most clients will send that cookie with each and every subsequent request. The most important thing to remember about cookies is that you *cannot trust* the data. When a cookie is sent to a client, it will be stored in plain text on that computer or device. Users can edit cookies as they please, or add and remove cookies, very easily. This makes incoming cookie data about as trustworthy as data that arrives on the URL with a `GET` request.

To put that a little more plainly: *do not trust cookie data*.

How do users edit their data? Well, there are a couple of options. First of all, let's look at using cookies with Curl. We can capture cookies into a "cookie jar" by using the `-c` switch. Take a look at what a well known site like amazon.com sets for a new visitor:

```
curl -c cookies.txt http://www.amazon.com/
```

The cookie jar file that was saved will look something like this:

```
# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This file was generated by libcurl! Edit at your own risk.

.amazon.com TRUE  /  FALSE  1355305311  skin            noskin
.amazon.com TRUE  /  FALSE  2082787201  session-id-time  2082787201l
.amazon.com TRUE  /  FALSE  2082787201  session-id       000-0000000-0000000
```

The format here contains the following elements:

- Domain the cookie is valid for

- Whether it is valid for all machines on this domain (usually TRUE)
- Path within the domain that this cookie is valid for
- Whether this cookie is only to be sent over a secure connection
- When this cookie will expire
- Name of the cookie
- Value of the cookie

Note the phrase "Edit at your own risk," which translates to developers as "Edit, and interesting things may happen." Whether working with a browser or Curl, it is possible to change these values wherever the cookies are stored, and they will be sent back to the server with a later request. With Curl, change the `-c` switch to a `-b` switch to send the cookies back with a request (use them both together to also capture incoming ones back into the file).

In the browser, your options will vary depending on which browser you use, but all of the modern browsers have developer tools either built in or available via a plug-in that enables you to see and to change the cookies that are being sent, as was mentioned in "Browser Tools" on page 6.

Working with Cookies in PHP

Cookies are key/value pairs, as I've mentioned, that are sent to the browser along with some other information, such as which paths the cookie is valid for and when it expires. Since PHP is designed to solve "the Web problem," it has some great features for working with cookies. To set a cookie, use a function helpfully called `setcookie()`:

```
<?php

setcookie("visited", true);
```

We can use this approach to show a welcome message to a visitor when he first comes to the site—because without any previous cookies, he won't have the "visited" cookie set. Once he has received one response from this server, his "visited" cookie will be seen on future requests. In PHP, cookies that arrived with a request can be found in the `$_COOKIE` superglobal variable. It is an array containing the keys and values of the cookies that were sent with the request.

When working with APIs, the same facilities are available to us. When PHP is a server, the techniques of using `setcookie` and checking for values in `$_COOKIE` are all that are needed, exactly like when we are working with a standard web application. When consuming external services in PHP, it is possible to send cookie headers with our requests in the usual way.

There's some nice support for sending cookies in PHP's `curl` extension, which has a specific flag for setting cookies rather than just adding headers. With PHP's `curl` extension, it is possible to do something like this:

```
<?php

$url = "http://requestb.in/example";

$ch = curl_init($url);

curl_setopt($ch, CURLOPT_COOKIE, "visited=true");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

A selection of other options can be set using cookies, as seen when we discussed capturing them into the cookie jar in the code examples in "Cookie Mechanics" on page 29. The expiry date is probably the most-used setting. The expiry information is used to let the client know how long this cookie is valid for. After this time, the cookie will expire and not be sent with any later requests. This relies on the client and server clocks being vaguely in sync, which is often not the case. Having exactly matching clocks is rare, and in some cases clients can have their clocks set incorrectly by a number of years, so beware.

The expiry can be set in the past to *delete* a cookie that is no longer needed. If an expiry has not been set for a cookie, it becomes a "session cookie," which means that it is valid until the user closes the browser. This is why you should always close your browser in an Internet cafe, even after logging out of your accounts.

Don't confuse the "session cookie" with the cookies PHP uses to track user sessions within a web application. You can use traditional PHP sessions in a web service, but it is unusual to do so—usually API requests are more self-contained and stateless than their web equivalents.

CHAPTER 5
JSON

JSON stands for JavaScript Object Notation, but don't be fooled by the name. Although it sounds as if it's a JavaScript-specific format, it is easily readable and writeable by a wide range of scripting languages today. It's a very simple, lightweight format, which can represent nested, structured data.

For example, if there were a data set that looked like this:

- message
  - — en: "hello friend"
  - — es: "hola amigo"

In JSON, that data would look like this:

```
{"message":{"en":"hello friend","es":"hola amigo"}}
```

If a piece of data is represented by a scalar value, then it is presented plainly. If it is structured (as shown in the previous example), such as an associative array or an object with properties in PHP, a curly brace is used to indicate a new level of depth in the data structure. The keys and values are separated by colons, and each record at a given level is separated with a comma.

It is also possible to show a list of items quite elegantly using JSON. Take this imaginary shopping list:

- eggs
- bread
- milk
- bananas
- bacon

- cheese

A JSON representation of this would simply be:

```
["eggs","bread","milk","bananas","bacon","cheese"]
```

As you can see here, many of the keys in the previous example are optional, and multiple values are enclosed with the simple square brackets. If this list was in fact the value of a property, then both kinds of brackets would be seen:

```
{"list":["eggs","bread","milk","bananas","bacon","cheese"]}
```

This example shows that our data contained a key/value pair, with the key "list."

When to Choose JSON

JSON gives a very clear indication of the original data structure and conveys the values within, but doesn't give us any specific information about the exact data types that were originally in use. Often, this isn't important; HTTP is entirely string-based anyway so it is usual to deal with this type of data in web-based applications.

JSON's strongest point is that it a simple data format. It doesn't take much storage space in comparison to XML and isn't too large to transfer "over the wire" or, in the case of mobile applications, over a potentially slow and patchy data connection! Since it is quite small and simple, it is inexpensive in processor terms to decode the JSON format, which makes it ideal for less powerful devices such as phones.

Use JSON when information about the exact data format isn't critical, and the effort needed to decode it must stay light. It's great for casual web or mobile applications—and of course it's absolutely ideal if you are supplying data to a JavaScript consumer, since it handles this data format natively and quickly.

Content negotiation over HTTP using headers has already been covered earlier in the book (see Chapter 3); this is how it is ascertained that the client would like a JSON response format. As an example, here are the headers for a request/response pair in which the consumer is requesting JSON and the API provides exactly that:

```
> GET /header.php HTTP/1.1
> Accept: application/json, text/html;=0.5

< HTTP/1.1 200 OK
< Content-Type: application/json

{"message":"hello there"}
```

You can see that the final entry in the example is the body of the response. The format of this is the same JSON that was covered earlier in this chapter. Setting the headers correctly is absolutely key, since without the correct `Content-Type` header, any application receiving this request will not know how to decode it. If it requested JSON,

it might *hope* that's what was returned, but the `Content-Type` should always match. If it isn't specified, many web servers will default to sending a `Content-Type` of "text/html", which is not only inaccurate, but also dangerous because a browser will try to display the content as HTML and allow embedded JavaScript—so do take care to set those headers correctly.

Handling JSON with PHP

This is very simple, which is another reason to choose JSON as a preferred output format! In PHP, you can use `json_encode()` to turn either an array or an object into valid JSON.

For example, the previous example showed some JSON that looked like this:

```
{"message":"hello you"}
```

To generate that from PHP (which is exactly how I generated the previous examples), I simply used this line:

```
echo json_encode(array("message" => "hello you"));
```

This shows a very simple array wrapped in `json_encode()` and using `echo` to output it so I can see it when I request the page.

To handle incoming JSON data and turn it into a structure you can use, simply use `json_decode()`, passing the string containing the JSON as the first argument. Sticking with our existing simple example, the code could look something like this:

```
$data = json_decode('{"message":"hello you"}');
var_dump($data);
```

This example includes `var_dump()` to show *exactly* what actually happens when the `json_decode()` function is used: by default, an object is returned. Here's the output of that script:

```
object(stdClass)#1 (1) {
  ["message"]=>
  string(9) "hello you"
}
```

Because there is no data-type information, JSON cannot tell whether this was an array with keys and values, or an object with properties, before it was turned into JSON; there is no difference between the two. We would get identical output from a script that looked like this instead:

```
$obj = new stdClass();
$obj->message = "hello you";
echo json_encode($obj) . "\n";
```

Similarly, the same output would be shown if an object of any other class were used; the object-type information just isn't included in JSON so it can't be retrieved at the other end. When calling the `json_decode()`, it is possible to convert the data to an associative array rather than an object—by passing *true* as the optional second argument:

```
$data = json_decode('{"message":"hello you"}', true);
var_dump($data);
```

This time around, our output is subtly different:

```
array(1) {
  ["message"]=>
  string(9) "hello you"
}
```

Whether you choose to work with objects or arrays is up to you, and really depends on the application and also the language. PHP objects are a little bit heavier than arrays, particularly in older versions of PHP (PHP 5.3 and earlier), so you will sometimes see better performance when using simple arrays for simple data.

JSON in Existing APIs

As an example of working with an API that uses JSON, let's take a look at a little piece of the GitHub API and use JSON for our examples. The examples here work with *gists*, which are similar to "pastebins"—places where you can put code or other text to share with others.

Our example is very simple; it creates a gist using PHP:

```
// grab the access token from an external file (to avoid oversharing)
require("github-creds.php");

$data = json_encode(array(
    'description' => 'Gist created by API',
    'public' => 'true',
    'files' => array(
        'text.txt' => array(
            'content' => 'Some riveting text'
        )
    )
));

$url = "https://api.github.com/gists";
$ch = curl_init($url);

curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_HTTPHEADER,
    array('Content-Type: application/json',
        'Authorization: token ' . $access_token)
);
```

```
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

There are a few things going on here that bear closer examination: sending JSON in requests, working with an Authorization header, and using credentials to gain access. You will notice that a variable `$access_token` is referenced, which isn't set in the code. This is set in the *github-creds.php* file, kept separate to stop access keys being leaked in this text. In a real development project, I'd still keep this separate, but for a different reason—using a separate file means I can exclude it from source control and avoid publicizing my access keys to the world! Of course it does happen, and if it does, you can always revoke your token and generate a new one. If you ever suspect that a token has been leaked, then do destroy it and generate another (something to bear in mind if your tokens are visible when demonstrating APIs).

A `POST` request is used to create a new gist (GitHub has a RESTful API—these examples will come up again in Chapter 8) and send JSON-formatted data along with it. In fact, this is a PHP array (because those are easy to understand and work with), which is then converted to JSON using `json_encode()`. The resulting output is given as the value for `CURLOPT_POSTFIELDS` and PHP sends it as the body of the request.

This example also sets some headers using the `CURLOPT_HTTPHEADER` option. The first one is `Content-Type`, which we have already seen in many examples, and the second one is `Authorization`. The Authorization header here includes the "token" and the access token within it, because the GitHub API uses OAuth2 for authorization. We discussed OAuth in Chapter 3.

If all goes well with the previous request, a 200 status code will arrive with the response and the new gist will be created. The gist will also be visible on the Web. Alternatively, the gist can be requested over the API: one of the things included in the response when requesting the new gist is a link to it, so we can extend the example to also fetch the gist. Since this is a public gist, no authorization is needed and it is possible to just grab the data using `file_get_contents()`, then `json_decode()` it. Here's the previous example again, with a few more lines added to illustrate grabbing the gist that was created:

```
// grab the access token from an external file
require("github-creds.php");

$data = json_encode(array(
    'description' => 'Gist created by API',
    'public' => 'true',
    'files' => array(
        'text.txt' => array(
            'content' => 'Some riveting text'
        )
    )
));
```

```
$url = "https://api.github.com/gists";
$ch = curl_init($url);

curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_HTTPHEADER,
    array('Content-Type: application/json',
        'Authorization: token ' . $access_token)
);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);

$gist = json_decode($result, true);
if($gist) {
    echo file_get_contents($gist['url']);
}
```

You can easily try this yourself, or for an even simpler way to interact with the GitHub API, simply request all your own gists using `https://api.github.com/user/username/gists` and replacing `username` with your own GitHub username. Many APIs use JSON in a similar way to exchange information with consumers, and this chapter has covered how you can do that with PHP.

CHAPTER 6
XML

XML is another very common data format used with APIs, and should feel familiar to us as developers. Anyone who has spent much time with the Web will understand the "pointy brackets" style of XML and will be able to read it. XML is a rather verbose format; the additional punctuation and scope for attributes, character data, and nested tags can make for a slightly bigger data size than other formats.

XML has many more features than JSON, and can represent a great many more things. You'll see more of this in Chapter 7, where complex data types and namespaces will come into play. XML doesn't have to be complicated; simple data can also be easily represented, just as it is with JSON. Consider our shopping list again:

- eggs
- bread
- milk
- bananas
- bacon
- cheese

The XML representation of this list would be:

```
<?xml version="1.0"?>
<list>
  <item>eggs</item>
  <item>bread</item>
  <item>milk</item>
  <item>bananas</item>
  <item>bacon</item>
  <item>cheese</item>
</list>
```

Working with XML in PHP isn't as easy as working with JSON. To produce the previous example, the code in Example 6-1 was used.

Example 6-1. Example of working with XML

```
<?php

$list = array(
        "eggs",
        "bread",
        "milk",
        "bananas",
        "bacon",
        "cheese"
);

$xml = new SimpleXMLElement("<list />");
foreach($list as $item) {
    $xml->addChild("item", $item);
}

// for nice output
$dom = dom_import_simplexml($xml)->ownerDocument;
$dom->formatOutput = true;
echo $dom->saveXML();
```

The starting point is the array that will be our list, then a `SimpleXMLElement` object is instantiated with a root tag that forms the basis for the document. In XML, everything has to be in a tag so an `<item>` tag has been introduced in order to contain each list item.

The final block only makes the output prettier, which isn't usually important because XML is for machines, not for humans. To get the XML to convert from a `SimpleXMLElement` object, call the `asXML()` method on that object, which returns a string. The string, however is all on one line!

The previous example instead converted from `SimpleXMLElement` to `DOMElement`, and then grabbed the `DOMDocument` from that. Set the `formatOutput` to true, so when a call is made to `DOMDocument::saveXML()` (to ask it to return the XML as a string), the resulting output will be nicely formatted.

When to Choose XML

XML's abilities to represent attributes, children, and character data all provide a more powerful and descriptive way to represent data than, for example, JSON. These same features make XML a great way to represent very detailed information, including data-type information, so it's a great choice when those details really do matter. It can include information about the types of data and custom data types, and each element can have attributes that cover even more information.

The larger data format is less of a concern when working with powerful machines and fast network connections, so XML is a popular choice when exchanging data between computers or servers, rather than sending things to phones or web browsers. Do be aware, however, that bandwidth costs may well still apply and may be a significant cost factor when large amounts of data are being transferred.

APIs are all about integration between systems and sometimes the choice of data format will be dictated by whatever is on the other end of the relationship. XML is particularly popular among many enterprise technology platforms such as Java, Oracle, and .NET, so users of these technologies will often request XML as a preferred format. If you are working with products or people that would prefer XML or are more confident handling this format, then offer XML, even if only as one of multiple data format options in your API.

XML in PHP

There are many ways we can work with XML in PHP, and they're all useful in different situations. There are three main approaches to choose from and they all have their advantages and disadvantages:

1. *SimpleXML* is the most approachable, and my personal favorite. It is easy to use and understand, is well documented, and provides a simple interface (as the name suggests) for getting the job done. SimpleXML does have some limitations, but it is recommended for most applications.
2. *DOM* is handy when a project encounters some of the limitations in SimpleXML. It's more powerful and therefore more complicated to use, but there are a small number of operations that can't be done with SimpleXML. There are built-in functions to allow conversion between these two formats, so it's very common to use a combination of both in applications, as we saw earlier in Example 6-1.
3. *XMLReader, XMLWriter, and their sister XMLParser* are lower-level ways of dealing with XML. In general, these tools are complicated and unintuitive but they have a *major* advantage: they don't load the entire XML document into memory at once. If very large data sets are involved, then this approach will be your friend.

XML in Existing APIs

There are a wide variety of APIs using XML. This next example looks at the photo-sharing site Flickr. The Flickr API provides a wide variety of functionality for working with photos, and every language will have some classes available that you can use with it, but there's no reason not to interact with the API directly. Example 6-2 shows how to find a list of kitten pictures.

Example 6-2. Fetching data from Flickr's XMLRPC service

```
<?php

require("api-key.php");
$animal = "kitten";
$data = file_get_contents('http://api.flickr.com/services/rest/?'
    . http_build_query(array(
        "method" => "flickr.photos.search",
        "api_key" => $api_key,
        "tags" => $animal,
        "format" => "xmlrpc",
        "per_page" => 6
    ))
);
```

This requests all the newest photos tagged "kitten" from Flickr. Flickr uses an API key passed as a URL parameter, which is a different approach to the Authorization header examples that have been demonstrated so far; each API will implement this in a different way. Although the header is a better practice, the developers of Flickr were trailblazers with implementing APIs for users, so there was no best practice when it was built. Since it's simply a GET request, this example uses `file_get_contents()` to fetch the carefully crafted URL. The resulting response looks something like this:

```
    <?xml version="1.0" encoding="utf-8" ?>
    <methodResponse>
        <params>
            <param>
                <value>
                    <string>
    &lt;photos page=&quot;1&quot; pages=&quot;131292&quot; perpage=&quot;6&quot; to-
    tal=&quot;787750&quot;&gt;
            &lt;photo id=&quot;8294579422&quot; owner=&quot;9482106@N04&quot; se-
    cret=&quot;9a3bac5af4&quot; server=&quot;8220&quot; farm=&quot;9&quot; ti-
    tle=&quot;Smokey 2012-12-18&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot;
    isfamily=&quot;0&quot; /&gt;
            &lt;photo id=&quot;8294535628&quot; owner=&quot;39066615@N08&quot; se-
    cret=&quot;90e31d5254&quot; server=&quot;8074&quot; farm=&quot;9&quot; ti-
    tle=&quot;Curious Tommy&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot; is-
    family=&quot;0&quot; /&gt;
            &lt;photo id=&quot;8293485771&quot; owner=&quot;28797694@N04&quot; se-
    cret=&quot;6650f1db57&quot; server=&quot;8213&quot; farm=&quot;9&quot; ti-
    tle=&quot;Tiny tooth&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot; isfami-
    ly=&quot;0&quot; /&gt;
            &lt;photo id=&quot;8294535494&quot; owner=&quot;39066615@N08&quot; se-
    cret=&quot;cc6fd4db0c&quot; server=&quot;8351&quot; farm=&quot;9&quot; ti-
    tle=&quot;Tommy&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot; isfami-
    ly=&quot;0&quot; /&gt;
            &lt;photo id=&quot;8294424628&quot; owner=&quot;26742588@N04&quot; se-
    cret=&quot;b6cd3f3556&quot; server=&quot;8224&quot; farm=&quot;9&quot; ti-
    tle=&quot;White Is The New&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot;
```

```
isfamily=&quot;0&quot; /&gt;
        &lt;photo id=&quot;8294402524&quot; owner=&quot;33892219@N06&quot; se-
cret=&quot;572968b650&quot;   server=&quot;8356&quot;   farm=&quot;9&quot;   ti-
tle=&quot;Cat Angel&quot; ispublic=&quot;1&quot; isfriend=&quot;0&quot; isfami-
ly=&quot;0&quot; /&gt;
&lt;/photos&gt;
                </string>
            </value>
        </param>
    </params>
</methodResponse>
```

Because the actual data is sent as an escaped XML string, the XML is parsed in PHP, then the string is extracted and parsed as a separate step in order to obtain the real data. Flickr doesn't supply the actual URL of the image, but gives enough information in the response that the instructions can be followed to assemble the actual URL. SimpleXML is used in this example—first to parse the response, then to parse the data inside it. This library represents child elements as object properties (and each child is a SimpleXMLElement), while attributes are accessed using array notation.

Here's Example 6-2 again, processing the data and outputting it with titles and `<img>` tags:

```
<?php

require("api-key.php");
$animal = "kitten";
$data = file_get_contents('http://api.flickr.com/services/rest/?'
    . http_build_query(array(
        "method" => "flickr.photos.search",
        "api_key" => $api_key,
        "tags" => $animal,
        "format" => "xmlrpc",
        "per_page" => 6
    ))
);

$simplexml = new SimpleXMLElement($data);
$data_array = $simplexml->params->param->value->children();

$photos = new SimpleXMLElement($data_array->string);

if($photos) {
    foreach($photos->photo as $photo) {
        echo $photo['title'] . "\n";
        echo '<img src="http://farm' . $photo['farm'] . '.staticflickr.com/'
            . $photo['server'] . '/' .$photo['id'] . '_' . $photo['secret']
            . '.jpg" /><br />' . "\n";
    }
}
```

The main body of the data contains a `<photos>` tag with multiple `<photo>` tags inside it—one for each photo. Each `<photo>` tag has some attributes inside it, so array notation is used to access these, retrieve the title, and build the image tag.

When working with APIs, different data formats are seen in use in a variety of settings. This chapter has shown how to create, work with, and parse XML. XML is more common on older and larger applications, but the data format will depend on the target market of the API, and many providers will offer multiple formats. Flickr, for example, offers the data in both JSON and XML format, but also offers a serialized PHP format. PHP's serialized format is very easy to work with and is a great choice for two PHP applications exchanging data; if you were to integrate Flickr into your own PHP application, this would be good format to choose. When integrating with applications on other technology platforms, XML is a better-supported choice.

CHAPTER 7
RPC and SOAP Services

In this chapter we'll be looking at two closely-related types of services: Remote Procedure Call (RPC) services, and SOAP. These two feel fairly similar, as they both involve calling functions and passing parameters, but their implementations are in stark contrast as the RPC is a very loose way of describing a service, whereas SOAP is very tightly specified.

RPC

RPC services quite literally *call procedures* (i.e., functions) *remotely*. These types of API will typically have a single endpoint, so all requests are made to the same URL. Each request will include the name of the function to call, and may include some parameters to pass to it. Working with RPC services should feel familiar to us as developers because we know how to call functions—we simply do so over HTTP.

To start out, consider Example 6-2 when a call was made to Flickr. The URL we made for that example was:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&tags=
kitten&format=xmlrpc
```

Within the URL, the name of the function can be seen in the "method" parameter (`flickr.photos.search`), the particular tags to search for are found in `tags=`, and the `format` parameter asks for the response in XML-RPC format.

There is a distinct difference between using an RPC-style service, with function names and parameters included in the data supplied, and having a service that is true XML-RPC, which is a very defined format. The option you choose depends entirely on the situation you and your application find yourselves in, but whichever it is, be sure to label it correctly.

Building an RPC service layer for an application can be achieved very simply by wrapping a class and exposing it over HTTP. Example 7-1 shows a very basic class that offers some toy functionality to use in the following examples.

Example 7-1. Example of the library class

```
<?php

class Library
{
    public function getDwarves() {
        $dwarves = array("Bashful", "Doc", "Dopey", "Grumpy", "Happy",
            "Sneezy", "Sleepy");
        return $dwarves;
    }

    public function greetUser($name) {
        return array("message" => "Hello, " . $name);
    }
}
```

To make this available via an RPC-style service, a simple wrapper can be written for it, which looks at the incoming parameters and calls the relevant function. You could use something along these lines:

```
<?php

include("library.php");
$lib = new Library();

if(isset($_GET['action'])) {
    switch($_GET['action']) {
    case "getDwarves":
        $data = $lib->getDwarves();
        break;
    case "greetUser":
        $data = $lib->greetUser(
            filter_input(INPUT_GET, 'name', FILTER_SANITIZE_STRING)
        );
        break;
    default:
        http_response_code(400);
        $data = array("error" => "bad request");
    }

    header("Content-Type: application/json");
    echo json_encode($data);
}
```

This example does a very simple switch-case on the incoming "action" parameter and passes in any variables as required (with validation, of course). We fetch the return data from the underlying library, then send the appropriate content negotiation headers and

the data, formatted as JSON. If the request isn't understood, then a 400 status code is returned along with some error information.

The previous example shows a very simple RPC-style service using JSON, and illustrates how easy it is to wrap an existing class of functionality and expose it over HTTP. Sometimes it's appropriate to use HTTP *within* an application to allow different components to be scaled independently; for example, moving comments to a separate storage area to be accessed by the original application rather than HTTP. In those scenarios, this approach of wrapping existing, hardened code can be very useful indeed, and is quick to implement.

Exactly as the difference between XML over an RPC service and XML-RPC is important to remember, the same applies here. The example shows JSON being returned by an RPC service, but JSON-RPC is something much more tightly specified. The *-RPC services can be a better choice when working with people or technologies that understand those and are happy implementing them. If the requirements are for something rather lighter and more approachable, then a simple custom format will work perfectly well. Standards are always good, especially for externally-available systems, but don't feel that they are your only choice.

SOAP

SOAP was once an acronym for Simple Object Access Protocol; however, this has been dropped and now it is just "SOAP." SOAP is an RPC-style service that communicates over a very tightly-specified format of XML. Since SOAP is well-specified when it follows WSDL conventions, little work is needed to implement it in an application, or to integrate against it; as of PHP 5.0, PHP has a really excellent set of SOAP libraries for both client and server.

You will see quite a few providers of SOAP implementations, and some open source tools such as SugarCRM and Magento also offer SOAP integration points. When looking at a new SOAP service, a tool called soapUI allows for browsing a service when a Web Service Description Language (WSDL) file is supplied. In fact, soapUI is excellent and can do about a hundred other things, including complicated functional testing, but for now we will look at its SOAP functionality.

As an example, I took the WSDL file from RadioReference and added it into soapUI, simply creating a new project, naming the project, and giving the URL to the WSDL file for this service. By default, this will create a request for each of the available methods, and generate an easy interface in which they can be executed. To run one, pick it from the list on the left, and then click the green Play button above the sample request. I used `getCountryList` as an example, as you can see in Figure 7-1.

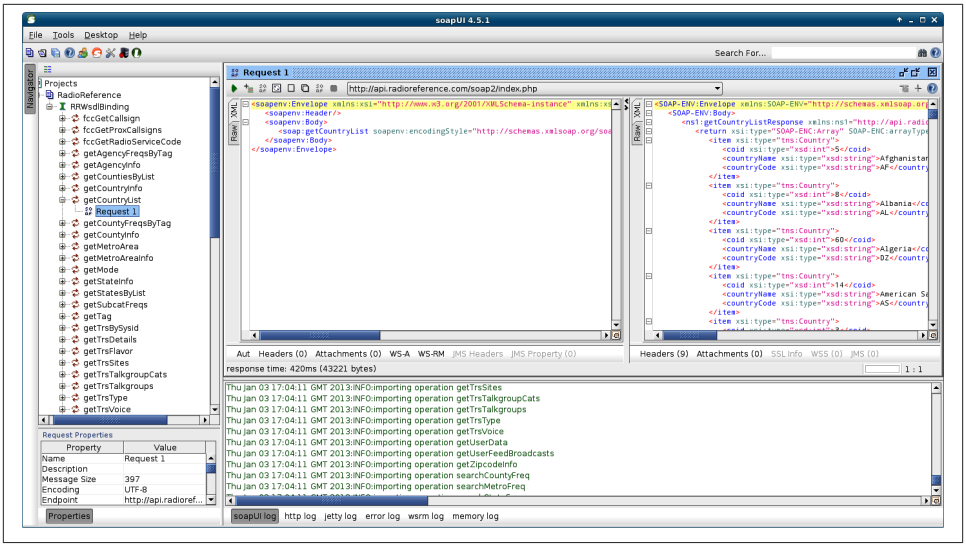


Figure 7-1. soapUI showing a request to getCountryList

The left half of the main pane shows the request that was sent, and the right half shows the response that was received. This gives a quick overview of how things look when using this API from our PHP code.

WSDL

This is a good moment to talk about the WSDL files that always seem to be mentioned whenever SOAP comes up. When it was first mentioned in this chapter, the acronym was defined as "Web Service Description Language," and this is a pretty good description of what is found in a WSDL file. It describes the location of a particular service, the data types that are used in it, and the methods, parameters, and return values that are available. The WSDL format is rather unfriendly XML, so it is best generated and parsed by machines rather than humans. If you do find yourself in the situation of needing to read one, it usually makes more sense to begin at the end of the document and then read upwards.

WDSL files are commonly used with SOAP, but they can be used with other types of web services. SOAP can also be used without a WSDL file, known in PHP as "non-WSDL mode." This chapter includes examples of SOAP with and without WSDLs, and an example of generating a WSDL file.

PHP SOAP Client

Returning to the countries list, we can acquire it from PHP quite easily using the SOAP extension. Take a look at this example, which does exactly that:

```
<?php

$client = new SoapClient('http://api.radioreference.com/soap2/?wsdl&v=latest');

$countries = $client->getCountryList();
var_dump($countries);
```

Simply using `var_dump()` doesn't create a very pretty output, but it does illustrate what these two lines of PHP have produced. The beginning of the output looks like this:

```
array(236) {
  [0]=>
  object(stdClass)#2 (3) {
    ["coid"]=>
    int(5)
    ["countryName"]=>
    string(11) "Afghanistan"
    ["countryCode"]=>
    string(2) "AF"
  }
  [1]=>
  object(stdClass)#3 (3) {
    ["coid"]=>
    int(8)
    ["countryName"]=>
    string(7) "Albania"
    ["countryCode"]=>
    string(2) "AL"
  }
  [2]=>
  object(stdClass)#4 (3) {
    ["coid"]=>
    int(60)
    ["countryName"]=>
    string(7) "Algeria"
    ["countryCode"]=>
    string(2) "DZ"
  }
```

Our two lines of PHP connected to a remote service and fetched us an array of objects containing the country information as requested. This shows the joy of SOAP, which is that very few lines of code are needed to exchange data between systems. The `SoapClient` class in PHP makes consuming data with a WSDL file trivial.

PHP SOAP Server

What about when we want to publish our own services? Well, PHP has a `SoapServer` that is almost as easy to use. Using the example library code from Example 7-1, we can make it available as a SOAP service in non-WSDL mode:

```
<?php

require('library.php');

$options = array("uri" => "http://localhost");

$server = new SoapServer(null, $options);
$server->setClass('Library');
$server->handle();
```

Since a WSDL is not used in the previous example, the Uniform Resource Identifier (URI) for the service must be provided. The example then creates the `SoapServer` and tells it which class holds the functionality it should expose. When the call to `handle()` is added, everything "just works." The PHP to call the code looks much like the previous example, but without a WSDL file, it is necessary to tell the `SoapClient` where to find the service by setting the `location` parameter and passing the URI:

```
<?php

$options = array("location" => "http://localhost/book/soap-server.php",
    "uri" => "http://localhost");

try {
    $client = new SoapClient(null, $options);
    $dwarves = $client->getDwarves();
    var_dump($dwarves);
} catch (SoapFault $e) {
    var_dump($e);
}
```

Again, just doing a `var_dump()` shows the results that are returned very clearly, but it isn't particularly pretty! The list of Dwarf names arrives in an array format:

```
array(7) { [0]=> string(7) "Bashful" [1]=> string(3) "Doc" [2]=> string(5)
"Dopey" [3]=> string(6) "Grumpy" [4]=> string(5) "Happy" [5]=> string(6) "Snee
zy" [6]=> string(6) "Sleepy" }
```

At this point, a working SOAP service exists, but not the WSDL file that is commonly used with it. The WSDL file holds the description of the service functionality, which means a file is created to describe our service, and should be recreated if any of the functions available change or if anything is added. Many technology stacks, such as Java and .NET, offer built-in functionality that makes it very easy to work with services that use WSDL files.

Generating a WSDL File from PHP

There are various solutions for generating a WSDL file from your library class code; some IDEs such as Eclipse have a button for it, and some frameworks also have this functionality. The examples here use a tool that will work regardless of the IDE or framework you use, because it's written in PHP: the php2wsdl tool.

To get set up, the files are extracted and placed in a *php2wsdl/* directory. Then a `WSDLCreator` object is instantiated and the files are added, along with information about which endpoint to use for which class, and a WSDL file is generated. Here's the code:

```
<?php

require("php2wsdl/WSDLCreator.php");

$wsdlgen = new WSDLCreator("LibraryWSDL", "http://localhost/book/wsdl");
$wsdlgen->addFile("library.php");
$wsdlgen->addURLToClass("Library", "http://localhost/book/soap-server.php");

$wsdlgen->createWSDL();
$wsdlgen->saveWSDL("wsdl");
```

This writes a file called *wsdl* to the local directory, and it contains the following:

```
<!--WSDL file generated by PHP WSDLCreator (http://www.protung.ro)-->
<definitions      name="LibraryWSDL"      targetNamespace="urn:LibraryWSDL"
xmlns:typens="urn:LibraryWSDL"    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"   xmlns:soapenc="http://sche
mas.xmlsoap.org/soap/encoding/"   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
   <message name="getDwarves"/>
   <message name="getDwarvesResponse"/>
   <message name="greetUser">
      <part name="name" type="xsd:anyType"/>
   </message>
   <message name="greetUserResponse"/>
   <portType name="LibraryPortType">
      <operation name="getDwarves">
         <input message="typens:getDwarves"/>
         <output message="typens:getDwarvesResponse"/>
      </operation>
      <operation name="greetUser">
         <input message="typens:greetUser"/>
         <output message="typens:greetUserResponse"/>
      </operation>
   </portType>
   <binding name="LibraryBinding" type="typens:LibraryPortType">
         <soap:binding  style="rpc"  transport="http://schemas.xmlsoap.org/soap/
http"/>
      <operation name="getDwarves">
         <soap:operation soapAction="urn:LibraryAction"/>
         <input>
                  <soap:body  namespace="urn:LibraryWSDL"  use="encoded"  encoding
Style="http://schemas.xmlsoap.org/soap/encoding/"/>
         </input>
         <output>
                  <soap:body  namespace="urn:LibraryWSDL"  use="encoded"  encoding
Style="http://schemas.xmlsoap.org/soap/encoding/"/>
         </output>
      </operation>
```

```
        <operation name="greetUser">
            <soap:operation soapAction="urn:LibraryAction"/>
            <input>
                    <soap:body namespace="urn:LibraryWSDL" use="encoded" encoding
Style="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                    <soap:body namespace="urn:LibraryWSDL" use="encoded" encoding
Style="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="LibraryWSDLService">
        <port name="LibraryPort" binding="typens:LibraryBinding">
            <soap:address location="http://localhost/book/soap-server.php"/>
        </port>
    </service>
</definitions>
```

The WSDL as it stands isn't terribly descriptive, as it can't guess what data types could be used or whether the methods should have arguments or return values. This is because PHP is dynamically typed, data types are not declared when defining variables or passing them into functions, and data types of return values are not declared either. Some other languages do declare data types and WSDL files usually contain detailed type information.

As an aside, look out for WSDL files with data types that PHP doesn't support—if the client or server is not in PHP, there can be a mismatch of formats in some cases. This is the main reason why so many WSDL files have fairly loose types, with strings rather than anything more specific. In fact, I have also seen an entire web service with a WSDL file that described a single method and accepted a custom XML format within it, for exactly this reason—not fun!

In order to make WSDL files more accurate, phpDocumentor comments can be added to our source code. Where the data types for parameters and return values are specified in documentation, the WSDL file will change to reflect the additional information.

PHP Client and Server with WSDL

Now there is a WSDL file to use with the `Library` example class, and the client and server code can be altered to take advantage of this. First, here's the server, with only the constructor needing to change:

```
<?php

require('library.php');

$server = new SoapServer("wsdl"); // wsdl file name
```

```
$server->setClass('Library');
$server->handle();
```

With the WSDL file in use, there's no need to give any other information. Just give the filename (this can be remote if appropriate) and all the location and other settings are picked up from there. The client can do exactly the same:

```
<?php

try {
    $client = new SoapClient("http://localhost/book/wsdl");
    $dwarves = $client->getDwarves();
    var_dump($dwarves);
} catch (SoapFault $e) {
    var_dump($e);
}
```

At this point, you're able to either build or consume RPC-style services in general, and XML-RPC, JSON-RPC, and SOAP in particular, with the use of handy tools such as soapUI.

CHAPTER 8
REST

REST stands for REpresentational State Transfer, and in contrast to protocols such as SOAP or XML-RPC, it is more a philosophy or a set of principles than a protocol in its own right. REST is a set of ideas about how data can be transferred elegantly, and although it's not tied to HTTP, it is discussed here in the context of HTTP. REST takes great advantage of the features of HTTP, so the earlier chapters covering this and the more detailed topics of headers and verbs can all come together to support a good knowledge of REST.

In a RESTful service, four HTTP verbs are used to provide a basic set of CRUD (Create, Read, Update, Delete) functionality: POST, GET, PUT and DELETE. It is also possible to see implementations of other verbs in RESTful services, such as PATCH to allow partial update of a record, but the basic four provide the platform of a RESTful service.

The operations are applied to *resources* in a system. The "Representational State Transfer" name is accurate; RESTful services deal in *transferring representations* of resources. A representation might be JSON or XML, or indeed anything else. So what is a resource? Well, everything is. Each individual data record in a system is a resource. At the first stage of API design, a starting point could be to consider each database row as an individual resource. Think of an imaginary blogging system as an example: resources might be posts, categories, and authors. Every resource has a URI, which is the unique identifier for the record.

A *collection* contains multiple resources (of the same type); usually this is a list of resources or the result of a search operation. A blog example might have a collection of posts, and another collection of posts limited to a particular category.

RESTful URLs

RESTful services are often thought of as "pretty URL" services, but there's more than prettiness to the structures used here. In Chapter 5, the GitHub API was used as an

example of an API using JSON; it is also a nice example of a RESTful API belonging to a system that developers may already be familiar with. Take a look at some of the URLs in this API:

- *https://api.github.com/users/lornajane/*
- *https://api.github.com/users/lornajane/repos*
- *https://api.github.com/users/lornajane/gists*

These delightful, descriptive URLs allow users to guess what will be found when visiting them, and to easily navigate around a predictable and clearly designed system. They describe what data will be found there, and what to expect. A key characteristic of RESTful URLs is that they only contain information about the resource or collection data—*there are no verbs* in these URLs. The best of API designs will have URLs that are "hackable"—that is to say that they are predictable enough to successfully guess where to find things. This links closely to the idea of hypermedia, which we'll discuss shortly.

In order to alter how a collection is viewed (for example, to add filtering or sorting to it), it is common to add query parameters to the URL, like so:

- *http://api.joind.in/v2.1/events* for all events
- *http://api.joind.in/v2.1/events?filter=past* for events that happened before today
- *http://api.joind.in/v2.1/events?filter=cfp* for events with a Call for Papers currently open

Notice that the URLs are *not* along the lines of `/events/sortBy/Past` or any other format that puts extra variables in the URL, but they use query variables instead. This data set, in both cases, still utilizes the `/events/` collection, but sorted and/or filtered accordingly.

Resource Structure and Hypermedia

Exactly how the resource is returned can vary hugely; REST doesn't dictate how to structure the representations sent. For example, a GitHub gist in JSON format looks like this:

```
{
  "created_at": "2012-12-20T15:37:51Z",
  "commits_url": "https://api.github.com/gists/4346013/commits",
  "description": "Gist created by API",
  "public": true,
  "html_url": "https://gist.github.com/4346013",
  "url": "https://api.github.com/gists/4346013",
  "forks_url": "https://api.github.com/gists/4346013/forks",
  "history": [
    {
```

```
      "change_status": {
        "additions": 1,
        "total": 1,
        "deletions": 0
      },
      "committed_at": "2012-12-20T15:37:51Z",
                                "url":     "https://api.github.com/gists/4346013/
f85e23d3443d0547292b202c3cd48881a28ebe9a",
      "version": "f85e23d3443d0547292b202c3cd48881a28ebe9a",
      "user": {
        "type": "User",
        "organizations_url": "https://api.github.com/users/lornajane/orgs",
        "gists_url": "https://api.github.com/users/lornajane/gists{/gist_id}",
        "followers_url": "https://api.github.com/users/lornajane/followers",
        "login": "lornajane",
        "events_url": "https://api.github.com/users/lornajane/events{/privacy}",
        "repos_url": "https://api.github.com/users/lornajane/repos",
        "following_url": "https://api.github.com/users/lornajane/following",
                "received_events_url":  "https://api.github.com/users/lornajane/
received_events",
        "gravatar_id": "372a6ef44baaa7291f1a6698348d2e98",
          "subscriptions_url": "https://api.github.com/users/lornajane/subscrip
tions",
         "starred_url": "https://api.github.com/users/lornajane/starred{/owner}{/
repo}",
        "url": "https://api.github.com/users/lornajane",
                              "avatar_url":    "https://secure.gravatar.com/avatar/
372a6ef44baaa7291f1a6698348d2e98?d=https://a248.e.akamai.net/assets.github.com
%2Fimages%2Fgravatars%2Fgravatar-user-420.png",
        "id": 172607
      }
    }
  ],
  "forks": [

  ],
  "user": {
    "type": "User",
    "organizations_url": "https://api.github.com/users/lornajane/orgs",
    "gists_url": "https://api.github.com/users/lornajane/gists{/gist_id}",
    "followers_url": "https://api.github.com/users/lornajane/followers",
    "login": "lornajane",
    "events_url": "https://api.github.com/users/lornajane/events{/privacy}",
    "repos_url": "https://api.github.com/users/lornajane/repos",
    "following_url": "https://api.github.com/users/lornajane/following",
              "received_events_url":    "https://api.github.com/users/lornajane/
received_events",
    "gravatar_id": "372a6ef44baaa7291f1a6698348d2e98",
    "subscriptions_url": "https://api.github.com/users/lornajane/subscriptions",
       "starred_url":  "https://api.github.com/users/lornajane/starred{/owner}{/
repo}",
    "url": "https://api.github.com/users/lornajane",
```

```
                        "avatar_url":      "https://secure.gravatar.com/avatar/
372a6ef44baaa7291f1a6698348d2e98?d=https://a248.e.akamai.net/assets.github.com
%2Fimages%2Fgravatars%2Fgravatar-user-420.png",
    "id": 172607
  },
  "updated_at": "2012-12-20T15:37:51Z",
  "git_pull_url": "https://gist.github.com/4346013.git",
  "comments": 0,
  "id": "4346013",
  "comments_url": "https://api.github.com/gists/4346013/comments",
  "files": {
    "text.txt": {
      "type": "text/plain",
      "filename": "text.txt",
      "size": 18,
      "language": null,
                                    "raw_url":      "https://gist.github.com/raw/
4346013/336516c8e23e55265245bf589ae56aafa9cbbcf2/text.txt",
      "content": "Some riveting text"
    }
  },
  "git_push_url": "https://gist.github.com/4346013.git"
}
```

Whereas a talk from Joind.in, also in JSON, would look like this:

```
{
    "talks": [
        {
            "talk_title": "Everything You Ever Wanted to Know About Deployment
But Were Afraid to Ask",
            "talk_description": "Deployment can be a real bugbear for many web
developers. From building something easy to deploy and manage; to coming up
with a repeatable, consistent process; to continuous deployment… deployment can
keep you up at night for months on end. In this talk I'll cover the following
topics:\n- The deployment maturity model\n- How to build a deployable applica
tion, from technology choice to instrumentation\n- Deployment velocity: Why
your process matters more than how often you deploy\n- Deployment tools and pro
cesses: How to automate your troubles away\n- CI/Automated testing: Know you're
deploying something good, or at least how worried you should be about it\n- Auto
mated testing vs monitoring: How they converge\n- When are you ready to deploy
continuously? How do you make the jump?",
            "start_date": "2012-11-08T13:00:00-05:00",
            "average_rating": 5,
            "comments_enabled": 1,
            "comment_count": 4,
            "speakers": [
                {
                    "speaker_name": "Laura Thomson",
                    "speaker_uri": "http://api.joind.in/v2.1/users/20041"
                }
            ],
            "tracks": [],
```

```
            "uri": "http://api.joind.in/v2.1/talks/7660",
            "verbose_uri": "http://api.joind.in/v2.1/talks/7660?verbose=yes",
            "website_uri": "http://joind.in/talk/view/7660",
            "comments_uri": "http://api.joind.in/v2.1/talks/7660/comments",
                  "verbose_comments_uri": "http://api.joind.in/v2.1/talks/7660/
comments?verbose=yes",
            "event_uri": "http://api.joind.in/v2.1/events/1056"
        }
    ],
    "meta": {
        "count": 1,
            "this_page": "http://api.joind.in/v2.1/talks/7660?start=0&resultsper
page=20"
    }
}
```

The two formats are quite different, and in fact the fields and formats available in a RESTful service will differ between each and every kind of service you could wish to encounter, but there are some common features, as can be seen even from this small sample size. Both responses include some nested information and some links out to other resources or collections.

The links to other resources/collections are called *hypermedia* and are an excellent inclusion in RESTful services; since every resource is identified by its URI, this data can be given as part of the response data. In this way, consuming clients can follow links, rather like a user clicking links on the Web, instead of assembling the next URL from the instructions and concatenating ID fields into it. Hypermedia makes the whole experience smoother and easier for consumers by offering the ability to find their way around easily. For example, using the previous data set, the following actions are available:

1. Look at this resource, and then visit the `comments_uri` to see the comments made on this talk.
2. See more information about the event this talk belongs to by visiting the `events_uri`.
3. From there, follow another piece of hypermedia in the `talks_uri` field to see a list of other talks at the event.

Another consideration when designing and working with RESTful APIs is whether or not it is useful to send additional nested data with the response to avoid a consumer having to make too many calls to the server. While GitHub and Joind.in both offer user information at their own locations, they also include some nested data in the responses shown here, which the consumer is likely to need.

On the other hand, sometimes too much information can lead to unnecessarily large amounts of data to transfer, and different APIs handle this in different ways. One

common pattern is that, by default, a subset of the information is returned, but functionality to retrieve more information is also offered—this is what the Joind.in `verbose_uri` offers. Alternatively, the extra information may be made available as a separate resource, such as offering `/article/42` as the data about a blog post, but excluding the (potentially large) body of the post, which can then be found at `/article/42/body`. Either approach shows consideration to the consumer, but which one is the right fit will depend on any particular scenario.

Data and Media Types

A RESTful service can offer a selection of data types, and it's very common to offer multiple types. Often, these will be JSON or XML, but there can be others; for example, Joind.in will respond to GET requests with an HTML data type if the Accept header requests it. The format decision will be made on the server, usually on the basis of the Accept header (you can read more about content negotiation in "Headers for Content Negotiation" on page 22).

Some services will allow a content indicator to be present in the URL itself, but this mixes up the identification of the resource with information about the representation desired. In general, the Accept header is the "right" way to indicate the preferred format, and a URL parameter may lower the barrier of entry, depending on your consumers.

Including a version number in your URL is a matter of taste. It is a very practical way to offer a service while identifying the current version of that service and opening the door to offering new versions of the service in the future. However, there are alternatives, and an elegant alternative is to use *media types*. These are invented content types that specifically describe the structure of the resource that will be returned, and can also include version information, so if the structure of a particular resource changes between versions, that change can be conveyed without a URL change.

Not all APIs will support media types, but they can be a good way to version representation structures for users who have a requirement to keep them predictable, and are happy to work with such specific content negotiations. GitHub does have some media type support (their reference page explains the detail very well) that goes beyond the usual `application/json` levels. They support media types specific to GitHub (`application/vnd.github+json`) and also support using the media type to specify the version of representation that should be returned (`application/vnd.github.v3`).

HTTP Features in REST

REST makes the most of HTTP's best features, placing all the metadata about the request and response into the headers, and reserving the main body of the communications for the actual content. This means that a correctly-implemented RESTful service will make

use of verbs, status codes, and headers so that all the extra information goes in the "envelope" of the request, and only the content is in the body. See Appendix A and Appendix B for tables of common status codes and headers. To look at how these various pieces go together, the next few sections take a walk through some examples of actual CRUD operations.

Create Resources

Resources are created by making a `POST` request to the collection to which the new resource will belong. The body of the request will contain a representation of the new resource, with the `Content-Type` header set appropriately so that the server will know how to understand it. When the resource has been successfully created, a successful status code will be included with the response.

It's common to choose a status code of 201 (which means "Created") when a new resource has been made, and to either return a representation of the new resource in the body, or to set a `Location` header, redirecting the consumer to the URI of the new record. It's perfectly valid to return a 200 ("Accepted, but not completed"), however, and helpful to return a representation of the resource (appropriately formatted according to the `Accept` header) including information about the URI of this new item.

In the event that the resource cannot be created, an informative status code and error message should be returned to the user. There is more in-depth discussion of error handling appropriate status codes in Chapter 12. In general, a 400 "Bad request" or 406 "Not acceptable" status code would be appropriate for a request that either wasn't understood, or didn't pass validation rules. There are also a very large number of other status codes to choose from (see Appendix A), depending on what exactly went wrong.

An alternative approach to using `POST` on a collection to create a new resource is appropriate in the situation when the consumer, rather than the server, sets the identifier of the new record. In this scenario, the representation of the new resource can instead be sent in a `PUT` request directly to the new URI. Care must be taken, when designing a system like this, to ensure that multiple consumers do not pick the same URIs, either causing conflicts or overwrites. At least make sure that these are dealt with in a sane way, perhaps using the 409 status code, which means "Conflict."

Read Records

To fetch representations of resources, use the `GET` verb applied to either a collection or an individual resource without sending any body content with the `GET` request. The resources will usually appear with exactly the same structure, regardless of whether they were requested within a collection or on their own. The status code will be 200 if the record(s) were successfully retrieved, although other "good" status codes may also be

used here such as 302 "Found" or 304 "Not modified" (more about caching in the next section when we discuss how to update records).

If, however, the record isn't successfully found, a status code describing the problem will be returned. In a vast number of cases, this will be a 404 status code, to indicate that the record wasn't found or doesn't exist. If the user isn't authenticated, a 401 "Not Authorized" status code may be returned; a user who has identified herself but doesn't have permissions to see this item may receive a 403 "Forbidden" instead. Any one of a number of other possible failure cases could also occur, and these should have the appropriate status codes associated with them.

If your API implements rate limiting, then it might be that the resource exists and the user has permission to see it, but she has exceeded her allotted number of requests in a given time frame. In this situation, either a 420 "Enhance your calm" or 429 "Too many requests" would be good statuses to return.

Some APIs (this includes GitHub) will return a 404 to indicate that the record exists but the requesting user does not have access to it. This makes it impossible to deduce the existence (or nonexistence) of a record without the rights to see it! Exposing such details is known as "leaking information" and in many settings it is something of which to be wary.

Update Records

To edit records RESTfully is a multistep process. First, the resource should be retrieved by `GET`. Then, the representation of the resource can be altered as needed, and that resource should be `PUT` back to its original URI. Even if only a small part of the record needs to be changed, REST deals with representations of resources so the whole resource will be fetched and sent back for the update. Identical to when a resource was created using `POST`, the `PUT` request will include the resource representation in the body and the appropriate `Content-Type` in the header.

It's quite common to include some identifying information for the contents of the resource, such as a `Last-Modified` header or an ETag, to allow for checking of whether the resource changed as a result of something else between the `GET` and `PUT`, as this isn't an atomic operation. This is closely linked to how cacheable different URIs are, which we'll cover later in this chapter (see "Caching Headers" on page 64).

For a newcomer to REST, updating a representation of a whole resource can seem cumbersome when only a tiny part of it is actually changing, but don't be tempted to diverge from this approach and break the RESTfulness of the design. If it really does seem like an alternative approach would be better, then you have two options: either create a subresource or use the `PATCH` verb.

Creating a sub-resource is simplest, if you want to change one field of a resource, and make that field available at its own URI. For example, if it seems like overkill to update a whole user record just to change an email address, then instead create a resource `/user/42/email`. This smaller resource can then be subject to `GET`, change, and `PUT` instead of fetching and then pushing back a whole user profile.

The alternative is to use `PATCH` to make a small change to an existing record. This isn't commonly-supported (support for PATCH is relatively uncommon even in modern APIs, but is also uncommon at an infrastructure layer, so beware that not all networks will allow PATCH), but an example is available because once again, GitHub has it. GitHub allows the user to make changes to individual fields, in a record by supplying the data you want to change and making a `PATCH` request instead of a `PUT` request to the existing resource's URI.

Delete Records

This is the most damaging move but it's also the simplest. The `DELETE` verb is sent with a request to the URI of the item to be deleted, with no body content necessary. Many services will return 200 for "OK"—or simply a 204 for "No content"—when an item was successfully deleted, and a 404 "Not found" if the item didn't exist. However, if the request was made to delete something, and the record doesn't exist, many services see that as "success" and will return 200 or 204, regardless of what really happened (unless the record couldn't be deleted for some reason, such as the user does not have the proper permission). This idea of always behaving in the same way each time the action is called is known as *idempotency* and is expected behavior for both `GET` and `DELETE` requests.

Additional Headers in RESTful Services

As seen in Chapter 3, it is possible to convey a wide selection of information using HTTP headers. There are a couple things that are relevant to the majority of RESTful services, which will now be examined in more detail: authorization and caching. These are both areas in which best practice for APIs differs very little from the best practices when building websites, but they bear revisiting in an API context.

Authorization Headers

A common header that has been seen earlier in this book is the Authorization header. This can be used with a variety of different techniques for authenticating users, all of which will be familiar to web developers.

The simplest approach to authorization is HTTP Basic authentication (for more detail, see the RFC), which requires the user to supply a username and password to identify himself. Since this approach is so widespread, it is well supported in most platforms, both client and server. Do beware, though, that these credentials can easily be inspected

and reused maliciously, so this approach is appropriate only on trusted networks or over SSL. When the user tries to access a protected resource using basic authentication, he will receive a 401 status code in response, which includes a `WWW-Authenticate` header with the value `Basic` followed by a *realm* for which to authenticate. As users, we see an unstyled pop up for username and password in our browser; this is basic authentication. When we supply the credentials, the client will combine them in the format `username:password` and Base64 encode the result before including it in the Authorization header of the request it makes.

Similar to basic authentication, but rather more secure, is HTTP Digest authentication (the Wikipedia page includes a great explanation with examples). This process combines the username and password with the realm, a client nonce (a nonce is a cryptographic term meaning "Number Used Once"), a server nonce, and other information, and hashes them before sending. It may sound complicated to implement but this standard is well understood and widely implemented by both clients and servers.

Other applications may have alternative approaches, including using cookies and sessions to record a user's information after he has supplied credentials to a login endpoint, for example. Others will implement solutions of their own making, and many of these will use a simple *API key* approach. In this approach, the user acquires a key, often via a web interface or other means, that she can use when accessing the API. A major advantage of this approach is that the keys can be deleted by either party, or can expire, removing the likelihood that they can be used with malicious intent. This is nicer than passing actual user credentials, as the details used can be changed. Sometimes API keys will be passed simply as a query parameter, but the Authorization header would also be an appropriate place for such information.

An even better solution has emerged in the last few years: OAuth (version 2 is much better than version 1). OAuth arises as a solution to a very specific and common problem: how do we allow a third party (such as an external application on a mobile device) to have secure access to a user's data? This problem is solved by establishing a three-way relationship, so that requests coming to the providing API from the third-party consumer have access to the user's data, but do not impersonate that user. For every combination of application and user, the external application will send the user to the providing API to confirm that she wants access to be granted. Once the relationship is established, the user can, at any time, visit the providing API (with which she originally had the relationship of trust) to revoke that access. Newer versions of OAuth are simple to implement but once again should *always* be used over SSL.

Caching Headers

Issues of caching are not specific to REST, or even to APIs, but they can help enormously when an API server needs to handle a lot of traffic. Requests that perform actions cannot be cached, as they must be processed by the server each time, but `GET` requests certainly

can be, in the right situation. Caching can either be done by the server, which makes a decision about whether to serve a previous version of a resource, or by clients storing the result of previous requests and allowing us to compare versions.

Giving version information along with a resource is a key ingredient in client-side caching, and also links with the non-atomic update procedures in REST as was mentioned in "Update Records" on page 62. When returning a resource, either an `ETag` (usually a hash of the representation itself) or a `Last-Modified` (the date this record last changed) is included with the response. Clients that understand these systems can then store these responses locally, and when making the same request again at a later point, they can tell us which version of a resource they already have. This is very similar to the way that web browsers cache assets such as stylesheets and images.

When a resource is served with an `ETag` header, some textual representation of the resource, perhaps a hash of the resource or a combination of file size and timestamp. When requesting the resource at a later date, the client can send an `If-None-Match` header with the value of the ETag in it. If the current version of the resource has a non-matching ETag, then the new resource will be returned with its `ETag` header. However if the ETag values do match, the server can simply respond with a 304 "Not modified" status code and an empty body, indicating to the client that it can use the version it already has without saving transferring the new version. This can help reduce server load and network bandwidth.

In exactly the same way, a resource that is sent with a `Last-Modified` header can be stored with that header information by the client. A subsequent request would then have an `If-Modified-Since` header, with the current `Last-Modified` value in it. The server compares the timestamp it receives with the last update to the resource, and again either serves the resource with new metadata, or with the much smaller 304 response.

RESTful versus Useful

REST is truly an elegant way to build services, and a nice way to work with data over HTTP. Not every application has requirements that are best met by a RESTful service, so don't be tempted to make architectural decisions based on the current fashionable technologies. Standards are always an excellent thing to follow; they've been created by people who have implemented this several times and learned from their mistakes. That said, don't be afraid to break the rules just as you would for any other architectural decision in software engineering. Many APIs are criticized because they are deemed "not RESTful." While I recommend that you follow the strategies in this chapter, it's acceptable for you to take inspiration from REST, rather than implementing it to the letter. Do make sure, though, that your API is still well documented, robust, and most of all: useful.

CHAPTER 9
Debugging Web Services

Anyone with extensive development experience has, by default, extensive debugging experience. Many of the skills learned as a PHP developer on more traditional web applications are very useful when working with APIs. Understanding how the pieces go together is probably the most important part of the puzzle. When you see a problem, you must answer several questions to determine the solution. Is that something that happened in the server? During transfer? Did the client not understand the response it received? And, if you have an application consuming an API, is it the application or the remote API with the problem? Narrowing down where exactly to look when things go wrong will save time and sanity.

Particularly with complex projects, it's easy for the problem of "it's not working" to be blamed on an API, especially if different teams take responsibility for different system components. In one such situation, the team providing the API created a requirement that all bug reports be provided with a replication case using only Curl. This caused grumbling from the developers of the consuming application, but at the end of the project, it emerged that half of the "bugs" in the API were in fact bugs in the consuming application code, and the developers had been able to track down and squash the bugs in their own system without interrupting the work of the API team. To this day I recommend Curl or other *very* simple replication cases, excluding as many other components as possible.

Whether Curl gives you a great example of how to replicate the bug or not, it is very useful. Re-running a request through Curl is very quick and very simple; most of the data to send can be stored in a file and shared between collaborators, giving everyone a very easy way to see the problem. Seeing the problem is half the challenge—actually fixing it is the other half.

If a team isn't familiar with Curl, or would prefer to work with GUI tools, then the same principle applies, but do try to keep as many dependencies out of the equation as possible. A bug that is impossible to replicate (or that seems complicated from its

description) probably won't get fixed quickly. In some scenarios, Curl won't do the job. For example, a system that uses OAuth1 would require much hashing of variables on the client side, which is very tricky to do with Curl, so in this case a different tool would be more appropriate.

Debug Output

Every PHP developer will have used `print_r()` or `var_dump()` at some point to return some additional information to the client during the course of the server processing a request. This technique is quick, easy, approachable, and can often be all that is needed to spot a typo or missing value.

When working with APIs, this can still sometimes be useful, but it does carry health warnings! If standard debug output is included with a response, and the client is expecting valid JSON, XML, or some other format, then your client will not be able to parse the response. For requests made from Curl, or in situations when the response is viewed as it is rather than parsed, the debug output technique is great. For other scenarios, other approaches may be a better fit.

Logging

When it is important to continue returning clean responses, more information can be acquired from an API, as it processes requests, by adding logging. This just means that, rather than sending debug information along with the output, it is sent somewhere else to be inspected (usually in a file on the server).

By default, PHP will write errors to the location specified in the configuration directive `error_log` in `php.ini`. If this is left empty, then PHP defaults to writing to Apache's error log (or stderr, if you're not using Apache). It is possible to write other information to this log, as well as the errors generated by PHP itself, using the `error_log()` function:

```
<?php

error_log("this is an error!");
```

Perhaps this looks like a rather oversimplified example, but at its most basic level this is all that is needed to add logging. When I look in the Apache error log on the server (the exact file location varies between platforms), I see this:

```
[Wed Dec 26 14:49:36 2012] [error] [client 127.0.0.1] this is an error!, refer-
er: http://localhost/book/
[Wed Dec 26 14:49:36 2012] [error] [client 127.0.0.1] File does not
exist: /var/www/favicon.ico
```

A couple of errors can be seen in the previous output. The first was sent by the code sample, which deliberately wrote a message to the error log, and the other is what

happened when my browser requested a favicon, but none existed. Using this approach, `error_log()` calls can be added into a project to help debug a particular issue. The output from the error log can then be checked to discover the additional information needed, rather than sending the additional error information back to the client.

Logging is a powerful technique; there are many more tricks available to make it even more effective. Log messages can be directed to a specific file, for example, rather than to the generic error log. To do this, use the `error_log()` function but with some additional arguments. The first argument is the message, as before, the second argument is where to send the message (`3` means "to a file;" for more detail see PHP's error log documentation), and the final argument is the file name to use:

```
<?php

error_log("all gone wrong", 3, "log.txt");
```

The file should be writeable by the user that the web server represents, and then the error message will appear in the file (beware: it doesn't add a new line after each message). Specifying a file means that all the debug information can be kept in one place and will be easy to follow. The file could be truncated between test runs to make it even clearer exactly what happened in any given scenario.

There are lots of excellent libraries around to make logging easier, and if you're using a framework, it will probably offer some great logging functionality. There are some great features in dedicated logging tools or modules that will help keep track of what's happening in your application without resorting to a `var_dump()` call in the middle of your JSON output:

Multiple storage options
: Many logging libraries support more ways to store log entries than just email or files. Usually it's possible to log into many different kinds of databases, various file formats, and other options. Depending on how you want to use the data, this can be very useful indeed.

Configurable logging levels
: Logging libraries usually allow you to state the level of error that is being logged; this is comparable to the PHP approach of having `ERROR`, `WARN`, `NOTICE`, and so on. The application allows you to set what level of logging should be performed. This means you can change the logging levels on a lower-traffic test platform when you want to see more detail, or increase them temporarily to see more detail during a particular set of operations. As a result, the log files don't become too huge when things are going well, but more detail can be obtained when required.

Sending error messages to email is very useful at times, particularly in situations when you want to immediately draw attention to the event. This might be because an event has occurred that must be reacted to urgently, such as a major and unusual failure case.

Another very useful situation when you might want to have your code email some details to you is if a bug can't be easily reproduced. Instead, set up your code to collect lots of information; should the bug manifest, then the application can let you know and send information alongside it. This gives more specific information about the problem than watching the general logs, and can be a great way to deal with elusive or edge-case bugs.

Debugging from Outside Your Application

Getting into your application to add debugging has all kinds of downsides. First, it requires that you have access to edit the code in both client and server, which often won't be the case. Making changes to your application to add debug information can cause the bug to disappear (this is known as a "heisenbug") or introduce other unintended effects.

A better alternative, particularly if you want to only observe the traffic arriving and leaving, is to use another application to inspect the traffic from outside the application. There are two tools in particular that can be used for this kind of approach: Wireshark and Charles. They offer fairly similar functionality in different ways, so the next sections cover them in turn.

Wireshark

Wireshark is a "network protocol analyzer." In plain English, that means that it takes a copy of the traffic going over your network card, and presents it to you in a human-readable way. You don't need to do any configuration of your application or network settings to use it; once it's installed, it can just start showing us the traffic.

When you run Wireshark, you see a screen like the one in Figure 9-1.

The lefthand column lets you pick which network card you want to capture (this screenshot is from my Ubuntu laptop; you'll see things a little differently on different operating systems). The "eth0" is your local wired network, "wlan0" is the wireless network, and "lo" is your local loopback. Look out for this if you're making API calls to localhost as they use "lo" rather than whatever connection your machine uses to access the outside world. If you're working with virtual machines, you will see more network connections here so you can pick the one for which you want to see the traffic.

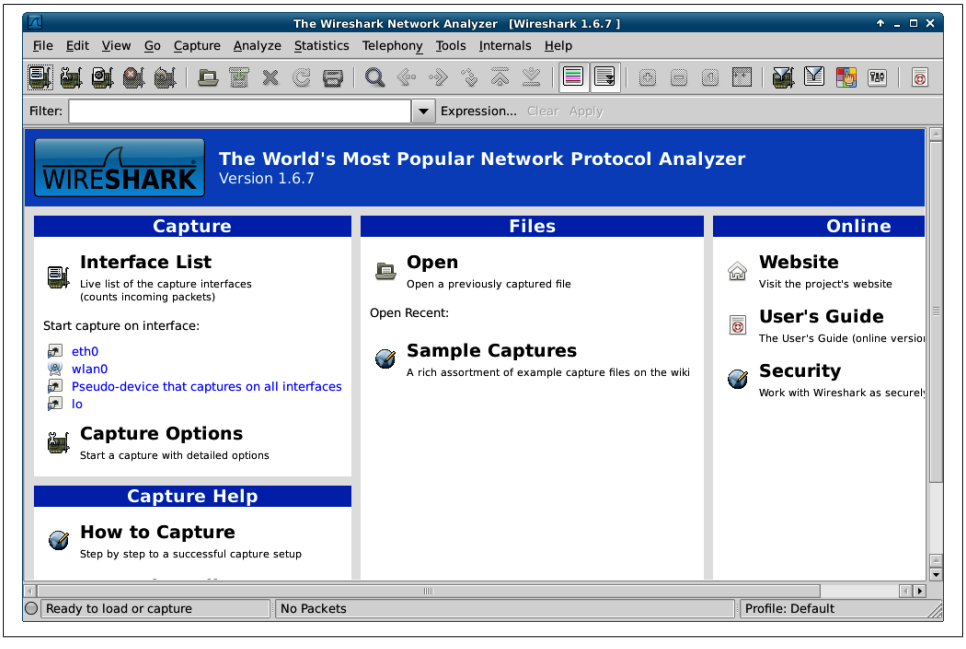


Figure 9-1. Initial screen when starting Wireshark

The other option you might want to use from this initial view is "open." Wireshark runs on your desktop or laptop and captures the traffic going over a network card on your machine. However, what if it's not your machine that you need the traffic from? It's rare to have a server with a GUI that you could install Wireshark on, so instead a command-line program called `tcpdump` (Windows users have a port called *WinDump*) can be used. This program captures network traffic, and the resulting files can be downloaded and opened in Wireshark to be analyzed.

Whether the traffic is captured live or comes from a file captured elsewhere, what happens next is the same: we view the traffic and start to examine what is happening. When I start a capture on my machine, I see something like Figure 9-2.

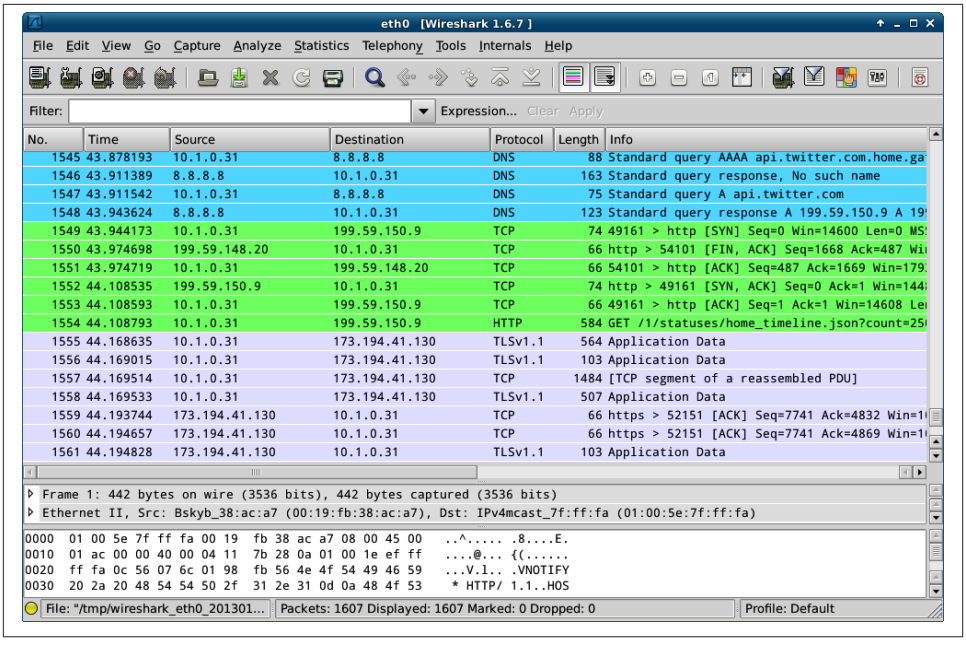


Figure 9-2. Wireshark showing all network card traffic

The first thing to do here is to restrict the amount of traffic being displayed to just the lines of interest by placing **http** in the *filter* field. Now a list of all the HTTP requests and responses that have been taking place are visible, making it possible to pick out the ones that are useful for solving a given problem.

Clicking on a request makes the detail pane open up, showing all the headers and the body of the request, or response, that was selected. This allows you to drill down and inspect all the various elements of both the body and the header of the HTTP traffic; when debugging, this is a very helpful technique for finding issues. Either nonsense is being sent by the client, or returned by the server, or (often) both.

To see the requests and responses linked together, right-click on either the request or the response and choose "follow TCP stream." With this, you can clearly see the requests and responses side-by-side, with the request shown in red (trust me that this is the first four lines, if you're seeing this in monochrome) and the response shown in blue in Figure 9-3.

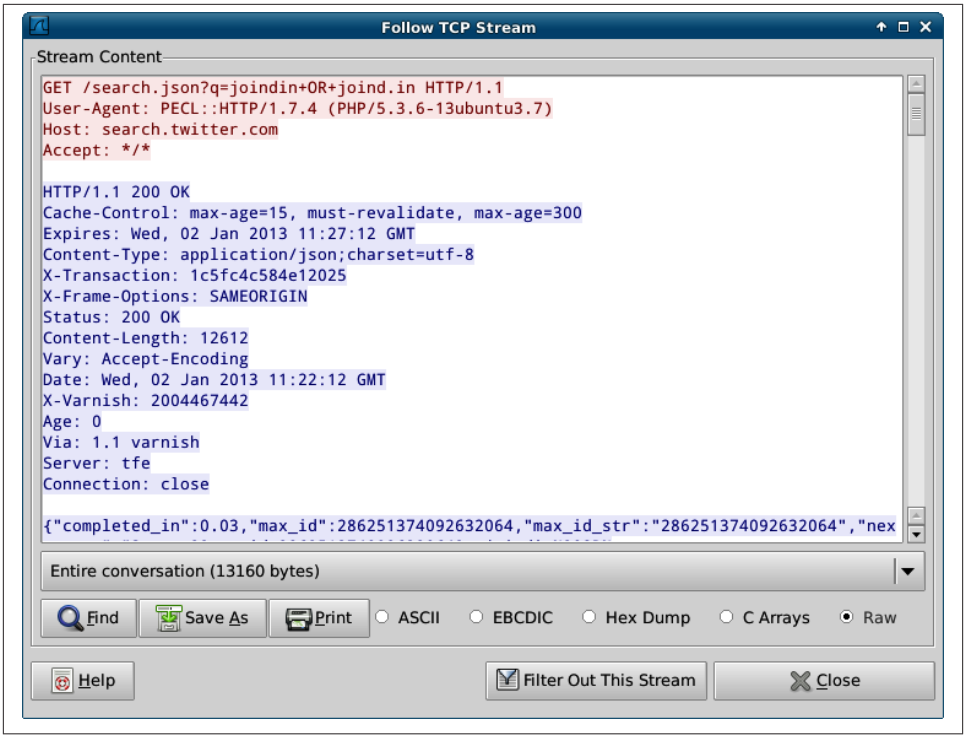


Figure 9-3. Wireshark showing a single TCP stream

The ability of Wireshark to quickly show what's going on at HTTP level without modifying the application is a huge advantage. Often, it's the first tool out of the box when something that "usually works" has suddenly stopped, and can show, for example, the `text/html` response, which is confusing the client that had been expecting JSON (a surprisingly frequent occurrence).

Charles Proxy

Charles is a paid-for product (a single license is $50 at the time of writing), but it's one that is absolutely invaluable, especially when working with mobile devices or when more advanced features are needed. Charles logs a list of requests and allows you to inspect them, just like Wireshark, but it works in quite a different way, as it is a true proxy, and requests are passed *through* Charles rather than the network traffic being duplicated.

Getting set up with Charles is straightforward; it automatically installs and will prompt you to install a plug-in for Firefox to enable proxying through Charles by default. If you're working with a web page making asynchronous requests, this is an excellent setup.

For those not using Firefox, you need to ask your application to proxy through Charles. Since it's common to have proxies in place, particularly on corporate networks, this is fairly easy to do on most devices; there are advanced settings when creating a network connection that will allow you to do this. You will need to enter the IP address of your machine, and the port number (8888 by default, but you can change it in the proxy settings in Charles) into the proxy settings fields when creating and editing the network settings. When a new device starts proxying through your machine, you'll get an alert from Charles that lets you allow or deny access.

Once everything is up and running, click on the "Sequence" tab and you'll see a screen similar to Figure 9-4.

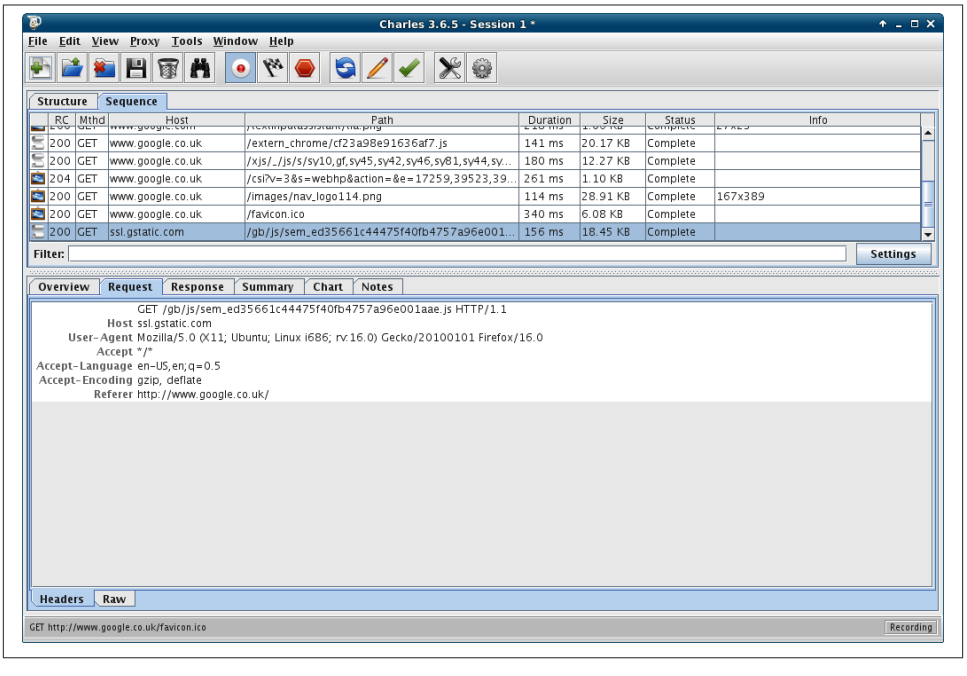


Figure 9-4. Charles showing some web requests in detail

The top part of the pane is a list of requests that came through the proxy (I just used Firefox to request *http://google.co.uk* as an example), and when you select one of these, the detail shows in the bottom pane. This area has tabs upon tabs, making all kinds of information available for inspection. There are the headers and body of the request and response, and if the response is valid JSON it will also decode it for you and show a nice representation.

If there's a particular response that allows you to observe a bug, you might like to *repeat* it; Charles makes this much easier than having to click around the same loop again to

replicate the bug. Simply locate the request you want in the top pane, and right-click on it to see "Repeat" in the context menu. This is really helpful for debugging, especially as you can export and import sessions from Charles, so you can pass this information around between team members.

Probably the nicest feature of Charles is its ability to show you SSL (Secure Socket Layer, or https) traffic without needing the private key from the server (which Wireshark requires). SSL is, by its very nature, not something than can be observed from the outside, so usually the result is something like the image in Figure 9-5. In simple terms, Charles creates its own certificates, and uses those to link with the browser. The "real" SSL certificate of the server is then used between Charles and the server. This setup is what is called "man-in-the-middle" and it's a common attack, which is why we have Certificate Authorities that offer trusted SSL certificates.

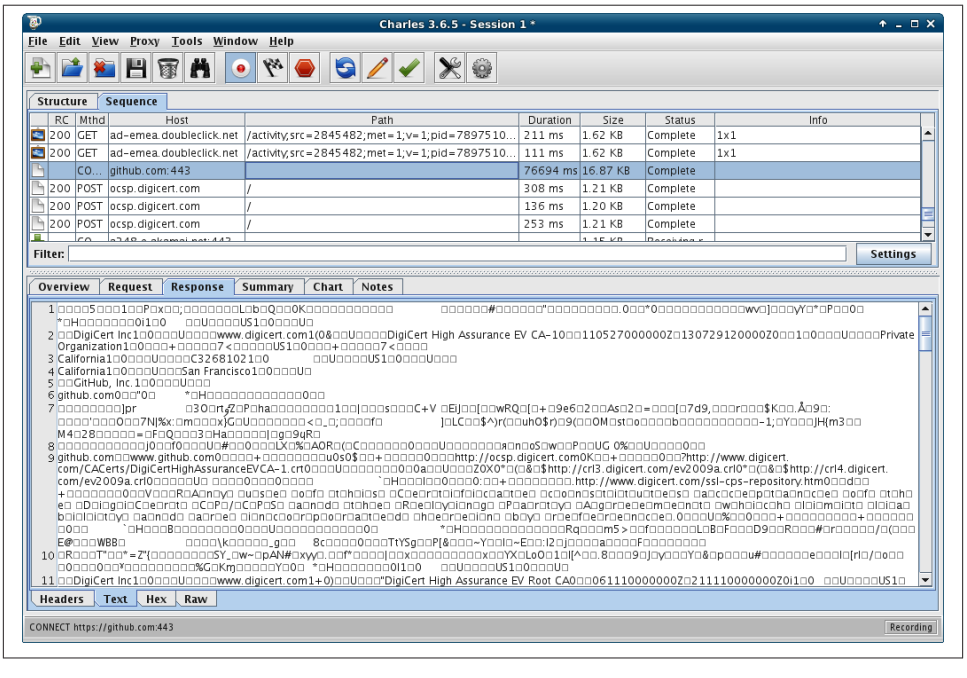


Figure 9-5. Charles showing https traffic without decrypting

You can enable SSL proxying on a per-site basis in Charles in the "SSL" tab of the proxy settings screen. When you try to access the SSL site now through Charles, you will see a warning that the SSL certificate is not trusted. Accept the warning and your application will operate as normal, with decoded content visible in Charles.

Two other features that are helpful with Charles are the ability to throttle traffic, and to rewrite any part of the request or the response as it comes through the proxy. Throttling

traffic allows you to simulate a selection of real-world network speeds, including 3G for a mobile phone. This is a key part of the development process, especially if your application and server are on a fast corporate network; the real world can look quite different! I will never forget testing games on phones in an underground car park to find out what happened when there was no reception.

The rewriting feature is also extremely handy—it makes it possible to change headers or bodies of requests or responses, restrict them to specific sites, and use regexes (see Figure 9-6). This can be handy for all kinds of reasons: dealing with server names that are different in your testing environment, trying out a new remote service, or testing whether a change of headers fixes a particular problem.

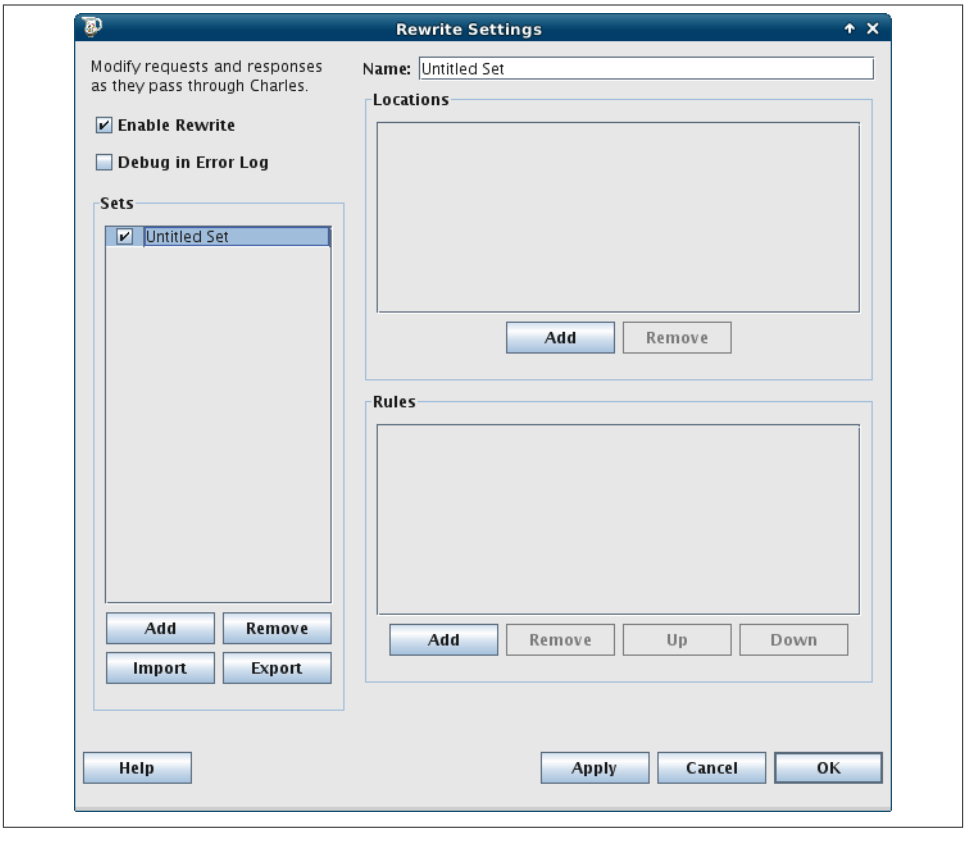


Figure 9-6. Charles allows rewriting of requests

Finding the Tool for the Job

Learning new tools costs time, even if the tools themselves don't cost money, and it can be hard to invest that time when there is pressure to fix the bug, ship the product, or close the deal. Getting to know as many tools as you can, however, and being able to set them up quickly, means you'll always be able to find the information you need. Think of it as an investment.

CHAPTER 10
Making Service Design Decisions

This is the million-dollar question: what kind of a service do I need for my next project? REST is cool, but RPC is familiar. JSON is lighter, but the client already works with XML. The API will be used by mobile consumers, or web consumers, or a reporting engine, or all of these.

There's rarely a clear-cut "one true way" when picking the best solution for a given API, but there are some key elements that can influence how to choose a solution that will be a good fit. API design is mostly engineering with a generous dash of common sense also required.

The big questions you need to ask at each step are these:

1. Who will be using this API?
2. What are they trying to achieve?
3. Which technologies do they use?

With these in mind, you can consider each of the following points.

1. It can be helpful to create some *user stories* to represent some of the expected users and tasks that the API will serve.
2. Building APIs is all about creating an interface point that makes sense when viewed from the outside, so the users' perspective is always the lens needed to scrutinize any decision.
3. Not every piece of data or possible piece of functionality in an application will necessarily make sense exposed over an API, so don't be tempted to build something huge immediately!

Service Type Decisions

The first decision to make when designing any API is one that can't be changed: decide what kind of a service you will offer. This depends on a combination of the audience and the type of service to be created.

For users who have larger systems using technology stacks such as Java, C++, or .NET, it may be easier for them to integrate with a SOAP service. This book covered SOAP in detail in Chapter 7, but basically it is an RPC-style service and is well-supported in PHP. Between platforms, problems can arise with mixed data types (such as when a data type exists in one of the languages but not in another), so do take care when picking data types.

For everyone else, a choice can be made that is based more on what kind of an API will be needed. If it will mostly be dealing with CRUD (Create, Read, Update, Delete) operations on data, then a REST API is a strong contender. It offers a simple way to work with data records and a well-designed RESTful API is very intuitive to pick up and use (read more about REST in Chapter 8).

The biggest pitfall with a RESTful API is the need to represent everything as a *resource*. This means that the URLs should be crafted to contain no verbs at all, and for an API that offers functional features, it can be quite tricky to reframe those ideas—both for the creators of the API and for those using it. These types of APIs have been quite popular in recent years but they won't be the right choice for every situation. The worst outcome here is to create a "RESTful" API, which isn't really RESTful.

Perhaps the simplest choice is one of the RPC formats. Developers on any platform are familiar with calling methods, passing parameters, and getting values returned. These services are easy to understand and will work well even for developers with limited API experience. RPC-style APIs are also very useful when using HTTP within existing applications, either to provide some modularity or to integrate two existing systems with a functional style, but they do have their downsides. Since the RPC services are usually made up entirely of `POST` requests, none of the responses can be cached by an HTTP proxy.

Consider Data Formats

A SOAP service will always use XML, but for RESTful or RPC services, the data format that fits best can be chosen. The most common options are JSON and XML, but there are also services that handle incoming form-encoded data formats, outgoing HTML formats, serialized PHP formats, YAML, and even plain text.

We saw in Chapter 6 some examples of XML being used with an RPC service, and SOAP is XML underneath. However, XML has plenty more applications than just SOAP, and can be used as the data format (or *a* data format) in any one of a number of different

styles of service. XML allows us to mark up elements with child elements, character data, and also attributes, but produces quite a large data size in return. Therefore, XML would do well when the bandwidth used for the transfers isn't slow or expensive, and the devices consuming the data have enough memory and processing power to handle and parse the data.

JSON is great for JavaScript applications, but they're not the only target market for this format. The majority of scripting languages have built-in support for JSON and will be able to serve and consume this format easily. JSON is also a great choice for mobile applications, where the smaller overall data size and simplicity of parsing the format are very useful for less powerful devices on potentially slow, patchy, or expensive connections.

HTML as a data format is an idea that isn't found in many textbooks, but certainly shows up in the real world on a regular basis. In its simplest form, we might return HTML in response to an AJAX request from a webpage, perhaps showing some new content in HTML on the page (something that you may already feature in your applications). It doesn't take a huge leap of faith from this to providing HTML as an optional output format for an API, if only for reading data. An example of this is found in the RESTful Joind.in API, where HTML is offered as an output format; if you request *http://api.joind.in* from your browser, the API reads your `Accept` headers and returns the data as HTML, with the hypermedia presented as clickable hyperlinks. This serves as excellent documentation for your service.

Accepting incoming requests from a web form, or in that format, can also be very web-friendly if the users of the API are mostly web developers and it is likely to be used mostly with or from a web page. This is a step away from the pure idea of exchanging data between machines, but can be a valuable option depending on the audience of the API.

If the user stories show that different consumers will want different data formats, then the API will need to return *multiple formats* such as XML, JSON, and perhaps HTML as well. This needs a bit of planning, but has major advantages because every consumer of your service will be able to ask for the data in the format that is right for their scenario. An application that takes care to make use of common templates or output handlers for each data format, used by every response sent, will be able to consistently return data in multiple formats.

Customizable Experiences

As well as choosing data formats, there are other variables for which the "right" choice to make will differ between the consumers of the API. An easy example is the number of entries you return. Returning all the data is fine…until the application becomes terribly popular, and suddenly the API is returning four thousand records instead of forty!

To improve this experience for everyone, APIs often offer pagination of data. As well as giving a way to specify which range of results to return, it is good practice to allow the number of results returned to be customized. A reporting server on a fast network might want all the data, whereas the mobile device with a patchy signal might only want the newest five records.

Another big variable is how much information to return with each request, and this decision usually manifests in two forms. When returning information about a particular item, should *all* the information be returned? And the follow up question: Should any related data be returned also? Including data means we'll sometimes be returning more information than needed, a bit like doing `SELECT * FROM …` in SQL. But if you omit data, then some consumers will have to make a large number of requests to obtain what they need.

Consider the example of the classic blog application. Should the API return the body of every article? If you're showing the user a list of articles, you probably don't want to show the entire text of the post, and to include all the text for all the articles would result in a huge response to send—but when showing an individual article, it will be an important piece of data to have. Allowing the consumer of your API to specify whether he needs headline data or detailed data, or offering different methods depending on whether a list of outline elements is needed or a single, in-depth method is required, will help users to get the best out of your API.

Now for the follow up question of whether to include related data. With the hypothetical blog post application, the post record itself will include, perhaps, the ID of the author. Your API will offer a way to fetch an author by his or her ID. But that means that when the consumer retrieves a list of articles, an additional call must be made for each of the items in the list to discover the name of the author so it can be displayed to the user, and these additional calls can be slow if many of them are needed. In a situation like this, it is quite clear-cut that we would return the name of the author with each article, to save lots of round trips to the server. In the real world, few situations are quite this clear-cut, and you will have to make some decisions about when data should be included and when it should be available separately. This is where the user stories I mentioned at the beginning of this chapter will help you to gain insight into what the "right" decision is, and in fact some resources should probably be made available in "brief" and "verbose" formats to allow consumers some choices.

Pick Your Defaults

It's important to offer users some choice, but also to offer a simpler path so that people can jump straight in and use your API without having to set up too many options. Every customizable option should have a default value that is returned if no preference is stated. Are you missing the `Accept` header? Send JSON. You don't have any pagination settings? Send the first 25 results. This approach allows people to get the best of the API very quickly and easily, and they can delve deeper to change the defaults if their requirements don't fit well with the defaults chosen.

Consider whether or not you will comply with all requests, though; if a consumer requests 1,000 results that might be expensive for your API to generate, you may still only send the first 200 (or whatever makes sense for your system). Similarly, some APIs will benefit from having rate limits. This means that each client can only make a certain number of requests in a given time period. Many APIs allow a very limited number of requests for unregistered users, and may allow differing levels of access to different customers, particularly for paid-for apps. Rate limiting is a way of making sure that you guarantee an expected level of service to all users by managing the load on your servers and allowing different users to have a level of access that suits them.

This philosophy of making things easy and useful to users, with minimal effort on their part, makes the barrier to entry much lower for your application and makes the experience of using a new API one of tolerance and welcome.

CHAPTER 11

Building a Robust Service

A robust service is one that feels secure and reliable to its users. Something that behaves unpredictably, sometimes gives incorrect results, and occasionally doesn't respond at all, is not what a consumer wants to integrate into her own applications. This chapter will look at what makes a robust service, and some techniques for making services as reliable and useful as they can be, both when things are going well and when they are not.

The best services exhibit consistent, predictable behaviors. This approach of having as much "sameness" as possible works well for consumers, who start to feel at home. As they use the service, they become familiar with how it will work, and will be able to find their way around and deal with any errors they encounter more easily. Most importantly, those consumers will be able to achieve their goals, which should give both consumer and provider a warm, fuzzy feeling.

Consistency Is Key

As PHP developers, we know only too well how difficult it is to use an interface that is inconsistent. The number of manual entries that use the words "needle" and "haystack" with very little correlation between which one should come first in any given situation (and one function where they can be passed in *either* order!) is our reminder of how painful this can be!

In our own applications, we can do better, but it is important to pay attention to the bigger picture and the existing elements of an API while working on building more features. In particular, consideration should be given to how things are named, how the parameters are passed in and returned, and what the expected behavior should be when something unexpected happens.

Consistent and Meaningful Naming

I recently worked with a system that had a function in it called `isSiteAdmin()`. Guess what it returned? Wrong! It actually returned the username of the current user, or `false`. There are plenty of examples of badly-named functions in the world, but please protect us from having any more to add to the list. Function names should be meaningful, and they should also be alike. So if there is something called `getCategories()` available, try to avoid adding a function called `fetchPosts()` or `getAllTags()` unless there's a good reason for the differences. Instead, fit in with the existing convention and call the functions `getPosts()` and `getTags()`.

The same applies to RESTful services, as well as those that contain function names, although it is slightly less of an issue when the clients are following hypermedia links. Look out for consistency in whether collection names are plural or not, for example.

Case-sensitive or not, make sure your service is absolutely case-consistent throughout.

The naming of parameters is also an area full of traps that are all too easy to fall into—and will annoy your users forever (or at least until you figure out how to release the next version without breaking their existing applications). The way that you name your parameters can give users a clue as to what they should be passing in. For example, a parameter called `user` is rather ambiguous but either `user_id` or `username` would help the user to send more accurate data through to your API.

Naming your parameters with "Hungarian notation" is probably a step too far, but aiming more at the verbose than the terse is probably in everyone's interests. If there's a field called `desc` then people will *probably* guess the correct meaning of the abbreviation from the context, but it is clearer to call the parameter `description` or `descending` or whatever it really means.

Common Validation Rules

The benefits of consistency were discussed already, but it is very easy to end up with slightly different validation rules for similar parameters in different settings; for example, whether extra address lines are optional or required between shipping and billing addresses. Also, try to avoid the irritatingly common situation of allowing a particular format of date/time information or telephone number in one place in your API, but not in another.

Make sure that incoming data is validated in the same way for the same kinds of data every time. An easy way to do this is to always use functionality that is built in, such as whatever your framework offers, or the fabulous Filter extension in PHP. Alternatively, and for types that are specific to your application, you can create a utility class that holds all the validations. In this way, you can add functions that check for particular kinds of data, and then reuse them across your application to ensure consistency.

Just like with the "needle" and "haystack" problems that are found in PHP, parameter ordering is important for RPC services. Figure out a plan to keep your service looking the same everywhere; does the API key need to be the first parameter, for example? Often it isn't obvious which parameters should be in which order, and in those cases it is best to simply pick something and then stick to it.

Predictable Structures

Structure of data is a key characteristic of a service, and a good API design will have it in mind when accepting requests, building responses, and also in the event of any error. APIs that return an array of results should *always* return an array of results. If there's one result, it still needs to be in an array. If there are no results, an empty array should convey this information. Suddenly returning `false`, or showing an item one level up from where it usually lives, is confusing; so take care to avoid it.

In most situations, the order in which parameters are provided, either as URL parameters or as part of body content, should not matter. Whether the parameter names or their values are case-sensitive can be made clear in the documentation; it is a challenge to keep these small details correct, particularly across a large API, but it is key and does greatly improve your system.

If an error should occur, it may well be the fault of the user. That said, the API ideally should help the user understand what went wrong and how the user can be better in their use of the API (because otherwise they will log a support ticket that you will have to fix). The entire next chapter is devoted to error handling, but at this point it seems important to mention that error responses should always be in a format, that is consistent across the API. If a user sees `not-success` in the status code that is returned with his response, he should immediately know how to get the information he needs about what went wrong, in a predictable format.

Predictability isn't just about data formats. Take care to follow patterns throughout an API regarding what happens when something is created, deleted, or not found.

Making Design Decisions for Robustness

Robustness is basically a measure of reassurance; how does the API behave both in good and bad situations? It can be tricky to know which design patterns are the best ones to follow, especially if you are new to APIs. In that situation, good advice would be to stick to the existing standards. These are well-known and understood, and will make it easier for people to integrate with your API or web service. Writing great documentation (see Chapter 13) is key to creating a great API; in general, anything without documentation will not be a good experience for anyone using it. Finally, always consider what should happen in the event that something goes wrong. Keep reading, as the next chapter is all about how to handle errors.

CHAPTER 12

Error Handling in APIs

Errors are a fact of life. Users will enter nonsense into your system, not because they are simpletons (although it does often look that way), but because their expectations and understanding are different from yours. The Internet is a very loosely-coupled affair and all kinds of things can and will go wrong at a technical level, once in a while. How your API handles these inevitable situations is a measure of the quality and design of your API, so this chapter gives some pointers on what to look out for and how to do it well.

Output Format

This is the golden rule: *always* respond in the format that the client was expecting. This means that it is never acceptable to return an HTML error message when the client expected JSON (in fact, in certain PHP versions, passing invalid JSON to `json_decode()` causes a segment fault!). If your system does return HTML messages when things go wrong, that is a bug and needs fixing. If an unexpected format is sent, the client will not be unable to understand the response and any error information contained in it.

In order to handle this requirement, there are some established patterns when designing our API that may help. Many modern applications have some kind of "front controller" pattern, in which all incoming requests are handled by a common entry point. This common front controller typically parses the request and figures out which part of the system it should be passed on to. We can put the same ideas into practice at the end of the request and make sure that the data to send back to the client always passes through a common point. At this point, it is possible to put in an output handler to correctly and consistently format the outgoing data correctly.

Here's a very simple front controller to give an idea of how this might look; you might use something along these lines, or follow the conventions of whichever framework you use:

```php
<?php

require "accept.php";

spl_autoload_register(function ($classname) {
    require ("inc/" . strtolower($classname) . ".php");
});

// create the correct view format
$accepted_formats = parseAcceptHeader();
$supported_formats = array("application/json", "application/xml");
foreach($accepted_formats as $format) {
    if(in_array($format, $supported_formats)) {
        // yay, use this format
        break;
    }
}

switch($format) {
    case "application/xml":
        $view = new XmlView();
        break;
    case "application/json":
    default:
        $view = new JsonView();
        break;
}


set_exception_handler(function ($exception) use ($view) {
    $data = array("message" => $exception->getMessage());
    if($exception->getCode()) {
        $view->status = $exception->getCode();
    } else {
        $view->status = 500;
    }
    $view->render($data);
});

// allowed controllers
$controllers = array("user", "post", "category");

// parse URL, first is class, then function
$pieces = explode('/', $_SERVER['PATH_INFO']);
if(in_array($pieces[1], $controllers)) {
    $classname = $pieces[1];
    $functionname = $pieces[2];

    $class = new $classname();
    $data = $class->$functionname();

    $view->render($data);
```

```
} else {
    throw new Exception("request not recognised", 400);
}
```

There's a lot happening here, so let's run through it in pieces. The first block is just the autoloader. To keep the example short, you will find all the classes in this example application are in `/inc` with lowercase file names matching their CamelCase class names. Next, the Accept header is parsed and the correct format is established using the example from the headers chapter. We work out the correct return format first, so it is then possible to return meaningful and understandable error information to a user in the event of any issues.

Next is the *exception handler*; this is a key pawn in our game of returning excellent and correctly-formatted content. Rather than passing error statuses and messages up and down potentially deep stacks, the application can be designed to use exceptions instead. Whenever something goes wrong, an exception will be passed up the stack and, if not handled, will arrive at the exception handler. This feature can be used to make sure that the only output from an error will be handled in the way the requesting client expects; care must be taken, however, to ensure that the messages returned by the exceptions are appropriate for public view—perhaps by always throwing specific exceptions with humane messages. A closure is used to bake the `$view` object into the exception handler, so that our output will be in the correct format.

Many applications follow patterns that separate out various pieces of functionality. You may have a system of routes and templates, where different URL patterns will be parsed to figure out which code to run. It's also very common to have an MVC (Model, View, Controller) style of application architecture; in an API this works very well, although the V (for view) becomes more like an output handler than a series of per-page templates. Here, we take the view that the first part of the URL will be the class name where the code can be found (in MVC, the "controller"), and the second part will be the function to actually run (in MVC, this is referred to as the "action"). When using a similar approach in your own code, do make sure that you are filtering incoming values correctly; the example uses the raw URL pieces for brevity, but a public-facing application would have much tighter security measures and would check that both class and function exist. Exactly the same principles apply to *all* incoming data; the URL, headers including cookies, and the `POST` variables (for example) are all to be treated with the same suspicion before using them in your code.

The biggest departure from a standard web architecture shows up right at the end of this front controller, where we send (sometimes called "dispatch") the request to the location where the code that can form the response is. Many of those controller-type patterns will then pass the program flow onto a template, and execution will end once the template has been rendered. In this example, those functions instead *return* data back to where the calls fan out from in order to be passed through a common output handler as the last step in the process.

The output handlers themselves can be beyond simple. To illustrate my point, here's the `JsonView` class used by the previous example code:

```
<?php

class JsonView
{
    public $status;

    public function render($data) {
        if($this->status) {
            http_response_code($this->status);
        }

        header('Content-Type: application/json; charset=utf8');
        echo json_encode($data);
        return true;
    }
}
```

The XML equivalent is slightly longer at 24 lines of code, but still not complex at all. The main things to remember are to set the correct `Content-Type` header and body format. Remember also that the status code needs to be set appropriately, depending on what action was performed.

Meaningful Error Messages

We all know how frustrating it is to get error messages from systems that say something like "an unknown error has occurred." This gives us absolutely no information at all on how we can coax the application to behave better. Even worse is an application I work with regularly, which will return the error message "Invalid permissions!" in the event that anything at all goes wrong, regardless of whether or not there is a problem with permissions. This leads to people looking in completely the wrong places for solutions and eventually filing very frustrated support tickets.

Error messages should be more than a tidy placeholder that the developer can use to find where in the code she should look when a bug is reported (there is also something to be said in favor of avoiding any copying and pasting of error messages for this reason). The information that an application returns in the event of an error is what lies between the application, the user, and the bug-reporting software. Anyone trying to use an application will have something he is trying to achieve and will be motivated to achieve that goal. If the application can return information about what exactly went wrong, then the user will adjust his attempts and try again, without bothering you. Users tend not to read documentation (developers in particular will usually only read instructions once something isn't working—all engineers do this), so the error information is what forms their experience of the system.

When something goes wrong, answer your user's questions:

- Was a parameter missing or invalid? Was there an unexpected parameter? (A typo can make these two questions arise together very regularly.)
- Was the incoming format invalid? Was it malformed or is it in a format the server does not accept?
- Was a function called that does not exist? (For common mistakes, you might even suggest what the user may have meant.)
- Does the system need to know who the user is before granting access? Or is this user authenticated but with insufficient privileges?

When it exists, give information about which fields are the problem, what is wrong with them, or if something is missing. It is also very helpful to users if you can *collate* the errors as much as possible. Sometimes, errors prevent us from proceeding any further with a request, but if, for example, one of the data fields isn't valid, we could check all the other data fields and return that information all at once. This saves the user from untangling one mistake only to trip straight over the next one, and also shows if the errors are related and could all be fixed in one go.

What to Do When You See Errors

Let us consider our other role in that relationship: that of the consumer of a service. Many of the APIs we work with are not ones we made ourselves, so inevitably we will be encountering some of the behaviors this chapter preaches against. What can we do when this happens? The best approach is to take baby steps.

First, go back to the last known good API call. At the very early stages of working with an API, that means reading the documentation or finding a tutorial to follow, and seeing if you can make any calls at all against this system. Some APIs offer what I call a "heartbeat" method and some offer a status page. Look for something that doesn't need authentication or any complicated parameters to call, and which will let you know that the API is actually working and the problem is at your end. Flickr has a particularly good example of this with their `flickr.test.echo` method.

Once it has been established that the target API is working, take a look at the call that was being attempted. Does it have any required parameters? Can the call be made in its simplest possible form, passing the smallest possible amount of data with the call? Even once things seem to be improving, it is advisable to approach changes to the API call in small steps, changing data format or adding a parameter, then checking that the response comes back as expected. Just like any kind of debugging, this iterative approach will help to pinpoint which change caused an error to occur.

While these test requests are being made, regardless of which tool is being used, take care to check the headers of the response as well as the body. Status codes, `Content-Type` headers, cache information, and all kinds of other snippets can be visible in the header and give clues about what is happening.

Do take the time to use the tools and tactics available to you, whether the errors are in your API or someone else's. In particular, the techniques covered in Chapter 9 will be superbly useful in such scenarios.

CHAPTER 13
Documentation

Raise your hand if you like writing documentation. Now raise your hand if you like discovering that the API you are to integrate against is well-documented and has examples of applications similar to what you need it for. These two are in direct conflict since very often, developers don't enjoy writing documentation, and while they often don't read it either, good documentation will ease the path of developers into using your service rather than logging a support ticket, or just leaving and using your competitors' offerings.

Your API might be the best the world has ever seen, but without any supporting documentation, or with bad/inaccurate documentation, people won't be able to use it. In fact, without considering great documentation as part of your project, one could argue that you may as well save yourself even more time and not build the API either!

There are many types of documentation, and a great web service probably needs a bit of all of them. The following sections will look at the various kinds of documentation that are useful to accompany a web service and give some suggestions of tools you can use to generate and maintain these.

Overview Documentation

This is the welcoming committee of your API; it gets people over the threshold and gives them confidence that they are about to have a good time. The overview documentation will set the tone of the API and provide some pointers for where to find more detailed information. In general, it shows the style and layout of the API and states the protocol(s) that are available. There will probably be some simple examples of requests and responses for common operations to show off the headers and body formats that should be needed. Showing the HTTP for both requests and responses is very useful, because it means that anyone running into problems can fire up a debugger and compare their results with the examples shown.

This chapter will also cover how users can identify themselves to the system, if they need to. Many services will allow some public access, others will ask that users link an API key to their login information on a website. If users need to actually log in, this overview section will cover how to do this, and the method will be the same across all the various parts of the API. This might be a username and password, or an OAuth process to follow, again with clear examples (bonus points if you can manage a real working guest account they can try) showing which credentials go where, where to get any necessary tokens, or how to craft a URL to which they can forward a user.

Information about error states belongs here in the overview, since they will be the same throughout the application. If the error states in your system aren't consistent, then go and read Chapter 12 before reading any further. If you use error codes, provide information about where to find more information about what they mean. If there will be information in the status code or headers, it is helpful to mention it here for any consumers not realizing that they need to look beyond the body text (although this should also contain useful information). Alongside the information about errors, you may also like to include some support information.

API Documentation

In the RPC services, it is common for the entry points to the service to be contained in a single class, and hopefully that class will have inline code documentation. If it does, and especially if this service is for an internal or technical audience, it may be possible to generate API documentation using phpDocumentor and supply this as a reference to your users (see Figure 13-1). This describes all the methods and parameters in the underlying class, but the PHP SOAP extension, for example, simply provides a very lightweight wrapper, so the generated documentation for the API of that class may well be a very useful artifact to generate and share. Do take care, however, that you're not exposing any undesirable information—for example, implementation details within protected methods.

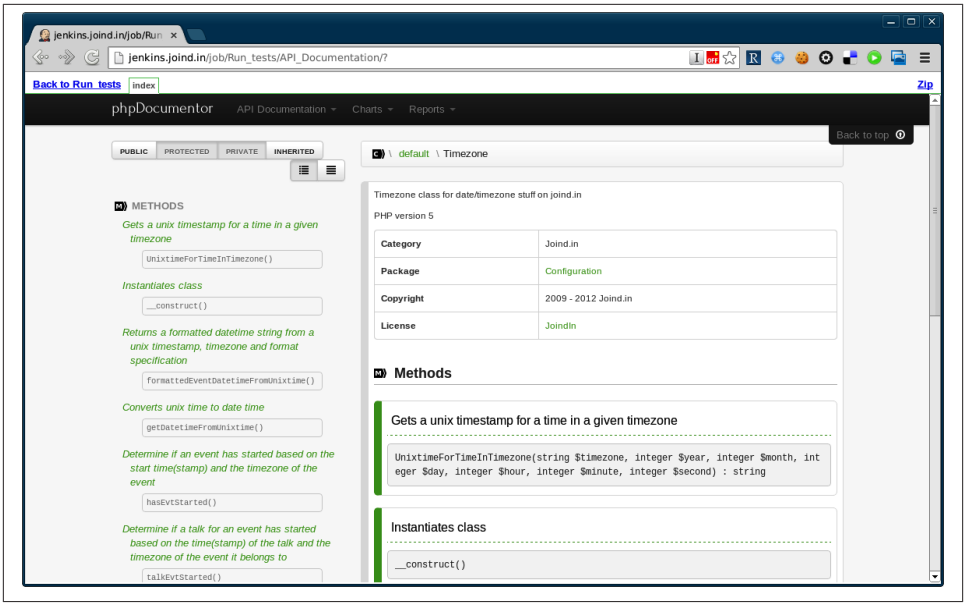


Figure 13-1. API documentation generated by phpDocumentor

Another way of documenting SOAP services is to supply a WSDL file, which was covered in Chapter 7.

For a RESTful service it is harder to generate documentation from our PHP code, but existing tools we have in our project can still be used and maintained alongside the API by linking our documentation to our other tools. One example is something like FRAPI, which allows you to configure a RESTful service for your application, and also generates stub documentation that you can expand upon.

Interactive Documentation

Some of the best documentation in existence for APIs allows a user to actually try out the request from the documentation page. One great example is Flickr, which offers an API Explorer that allows the user to enter data into the fields and then make the request from the online documentation itself (see Figure 13-2). This allows the user to try the feature as herself or as an anonymous user and set any of the available parameters for a particular method. Flickr gets extra points for technical merit, as they include some handy reference numbers, such as your own user ID and some recent photos uploaded to your account on the same page.

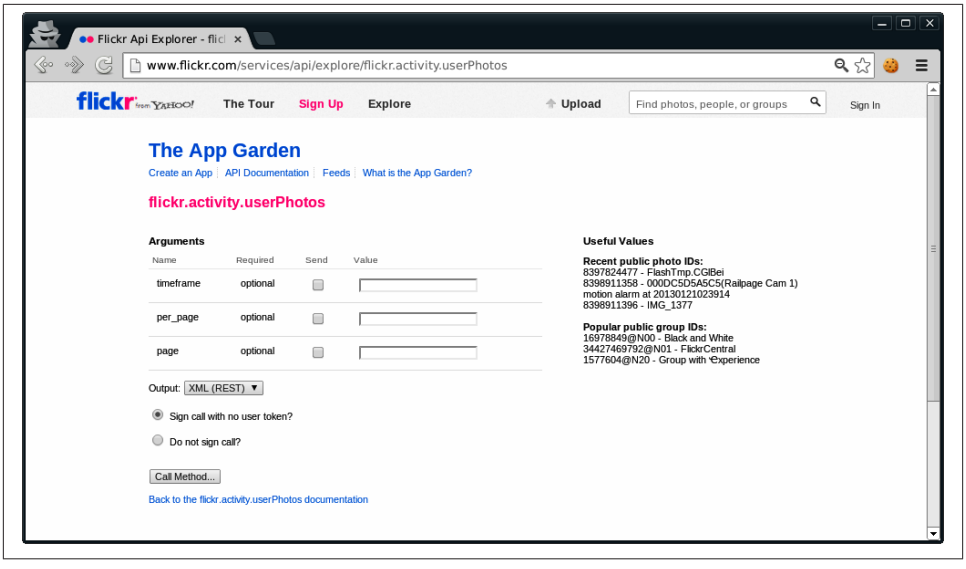


Figure 13-2. Flickr offers interactive API documentation

There are plenty of tools available to help create something similar for another project; alternatively, you could create a simple Web-based way to try your API that you can include with your documentation. For an existing system, you could do worse than the I/O Docs tool. It's written in Node.js and the code is available on GitHub, so you can amend it as you need to. You create a configuration file describing how your API can be used, which URLs can be called, what format and parameters to use, and so on. Once you are done, I/O Docs creates a page showing these available actions and parameters as a web form, and allows users to click the alluringly-named "Try it!" button to try making a request and viewing the response. This is used by a few online APIs; for example, Klout (Twitter metric tools) uses it to document its API, as you can see in Figure 13-3.

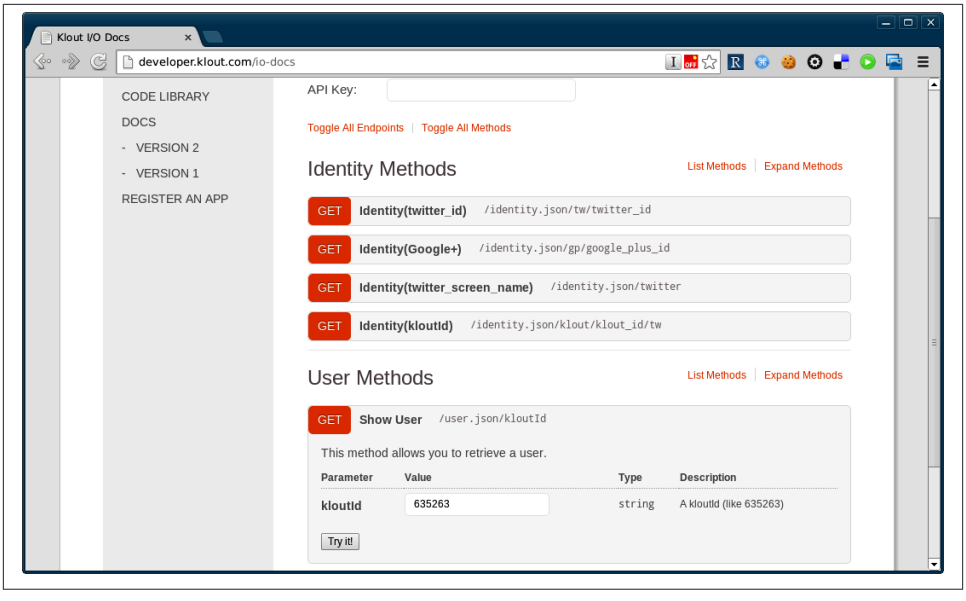


Figure 13-3. Klout uses I/O docs to create its interactive documentation

There are other great options such as the apiary.io site, and it's always worth looking out for new tools being released. Whichever route you take, interactive documentation serves as a quick way for a new or potential user to figure out whether she can use your API for her purposes, and what would be involved to do so.

Tutorials and the Wider Ecosystem

Documentation is about so much more than lists of accessible functionality. It is about showing how the API actually solves problems, and how it looks when it is used in the real world. A common criticism of software library documentation is that, while each function is documented, it can be very hard to know which function you want to use. Giving practical tutorial examples is really useful, even when they are not exactly what a user was looking for. Such examples can often give enough clues for a developer to piece together what he should do for his own application. If any of your users write on their own sites about how to use your tool, send them a T-shirt, or at least a tweet, and link to their material. That kind of content is hard to create, and having supportive outsiders shows what a great following your tool has.

Make sure your users know where they can go for support; then go and find where they actually *ask* for help. While you may set up user forums to help people with their queries and make those details public so that other people can find answers to common questions, users often don't follow the paths you set for them. Sometimes it is necessary to "pave the cowpaths" and follow where they lead. To this end, set up a search alert for

your product or application name with a search engine, and make sure that when questions do pop up in other places (such as StackOverflow), someone is able to respond.

Having documentation outside of your own control is a very positive thing, although it can feel a little frightening at first. Users are the word of mouth that spread influence, and often they can become your biggest advocates and very effectively help one another. Welcome those users and credit them where you can; documentation from any angle is a resource that's valuable to any project and it's vital for anything public. It is referred to as the "ecosystem" because it's the world your application exists in.

APPENDIX A
A Guide to Common Status Codes

This section outlines some of the most common status codes in use in HTTP APIs, their meaning, and some notes about when they can be used.

Code	Meaning	Notes
`100`	Continue	For a large request, a client can send just the headers and `Expect: 100-continue` as an additional header. If the 100 status is received in response, the client can then send the request as normal. Think of it as "go ahead"—in fact, many libraries will handle this for you and make the second request without further prompting.
`200`	OK	This is good news, everything worked as expected.
`201`	Created	A new resource was created. This is often accompanied by a `Location` header or a representation of the new resource in the body of the request.
`202`	Accepted	This is useful if something is taken to be actioned later, such as being placed on a queue for asynchronous processing.
`204`	No Content	The request was successful, but there is nothing to return. Perhaps this is the result of a `DELETE` request.
`301`	Moved Permanently	The content is at a new location, and this is a permanent change. Links to the old URL must be updated, and this change will often be cached for long periods.
`302`	Found	This is much like a `200`, but the content was not at the location specified. Usually this is seen when an application uses rewrite rules.
`304`	Not Modified	This is sent in response to a request that included information such as an `ETag` or `Last-Modified`, which indicates that the resource is cached and specified which version the client has. This status code means "use the one you have" and is useful to avoid repeatedly transferring large representations that don't change.
`400`	Bad Request	This is the general "something went wrong" status. Sometimes there may be no more detail to offer; at other times, you may choose not to transmit anything more.
`401`	Unauthorized	Credentials are needed in order to access this resource.
`403`	Forbidden	This contrasts with `401` and means that any credentials given were not sufficient to access this resource.

Code	Meaning	Notes
404	Not Found	A request was made for something the server doesn't have or doesn't know how to provide. Alternatively, a request was made for a resource that isn't available to this user and the 404 doesn't leak information about the potential existence of such a resource.
405	Method Not Allowed	The verb used to access this URL isn't supported—this is useful if, for example, you don't allow updates to a resource but a PUT request was received.
406	Not Acceptable	The server cannot generate a response in accordance with the `Accept` headers that came with the request.
409	Conflict	There is a mismatch between versions of resources, such as an incoming update when the resource has changed in the meantime.
410	Gone	A resource did exist, but doesn't any more. Many services will simply return a 404 here, or a 409 may also be appropriate, particularly if something is trying to perform an update on the resource.
415	Unsupported Media Type	The media type specified in the `Content-Type` header isn't understood by this server.
429	Too Many Requests	Usually used with rate-limiting schemes, although Twitter uses 420 "Enhance Your Calm" for this purpose.
500	Internal Server Error	An unhandled error occurred, and is the fault of the server rather than the client. In PHP applications, PHP has usually segfaulted, leaving the web server unable to return any useful information.
501	Not Implemented	The server can't handle this request; it may also indicate that a documented feature is currently still under construction.
502	Bad Gateway	This indicates that a proxy server of some sort has failed, such as a load balancer.
503	Service Unavailable	This is usually seen when a server is temporarily offline, such as during a planned maintenance window. Often, it really means "try again later" but it also discourages caching, and is particularly useful to stop search engines from finding and caching your temporary holding page.

For a full list of status codes, there is an excellent reference on Wikipedia.

APPENDIX B
Common HTTP Headers

Here we look at a series of often-used headers, whether they are request or response headers, and how they can be used.

Header	Request	Response	Notes
`Accept`	yes		This shows the formats, with an indication of preference, that the requesting client can understand. Closely related are the additional headers `Accept-Charset`, `Accept-Encoding`, and `Accept-Language`.
`Authorization`	yes		This is free-form information to prove a user's identity. This is used in basic authentication, digest authentication, OAuth, and so on; each has their own format of exactly what goes in the header.
`Cookie`	yes		Cookies are key/value pairs sent with each request, separated by a semicolon. This is the sister header to `Set-Cookie`.
`Content-Length`	yes	yes	Any request or response with body content should also have the `Content-Length` in bytes in the header; often your HTTP library will calculate this for you.
`Content-Type`	yes	yes	Any request or response with body content should include the `Content-Type` header to provide information about the format of that body content. As with the `Accept` headers, `Content-Encoding` and `Content-Language` may also be sent to give information about the format of the content.
`ETag`		yes	This is an identifier for the version of the resource that is being returned. If the client caches the resource, this information can be used with `If-None-Match` to work out whether a resource has been updated or if the previous version can be used.
`If-Modified-Since` and `If-None-Match`	yes		This informs the server that there is a cached copy of this resource and allows the server to return a 304 status code if that resource is still valid.
`Last-Modified`		yes	This provides information about when this resource was last updated; the client can use this to check if it has the most recent version of the resource upon subsequent requests.

Header	Request	Response	Notes
`Location`		yes	This provides information about a location and is used either with 300-series status codes when redirecting, or with 201/202 to give information about the location of a new resource.
`Set-Cookie`		yes	This sends cookies to be stored on the client and sent back in a `Cookie` header with later requests.
`User-Agent`	yes		This provides information about the client software making the request.

About the Author

Lorna Jane Mitchell is an independent web development consultant, specializing in PHP and APIs in particular. With over 10 years of PHP development experience across a wide variety of industries, Lorna learned many lessons the hard way and always has a story to tell. Lorna is also an experienced trainer, offering training to private clients around the world, and teaching public courses. A prolific writer, Lorna writes for a number of publications, and frequently for her own blog.

Colophon

The animal on the cover of *PHP Web Services* is an Alpine Accentor (*Prunella collaris*).

The cover image is from Wood's *Animate Creation*. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.